TURING 图灵电子与电气工程丛书



Microchip公司
全套课程解决方案推荐用书

32位单片机C语言编程

基于PIC32

Programming 32-bit Microcontrollers in C

Exploring the PIC32

[意] Lucio Di Jasio 著
张鼎 岳虹 等译





人民邮电出版社
POSTS & TELECOM PRESS

TURING 图灵电子与电气工程丛书

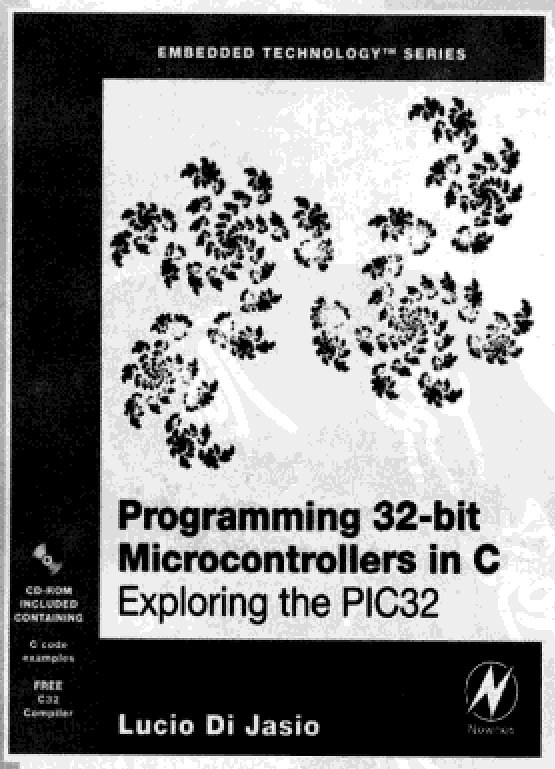
32位单片机C语言编程

基于PIC32

Programming 32-bit Microcontrollers in C

Exploring the PIC32

[意] Lucio Di Jasio 著
张 鼎 岳 虹 等译



人 民 邮 电 出 版 社
北 京

图书在版编目（CIP）数据

32位单片机C语言编程：基于PIC32/（意）贾西欧（Jasio, L. D.）著；张鼎等译. —北京：人民邮电出版社，2009.12

（图灵电子与电气工程丛书）

书名原文：Programming 32-bit Microcontrollers in C：Exploring the PIC32

ISBN 978-7-115-21612-0

I. ①3… II. ①贾… ②张… III. ①单片微型计算机－C语言－程序设计 IV. ①TP368.1 ②TP312

中国版本图书馆CIP数据核字（2009）第189969号

内 容 提 要

本书介绍32位单片机PIC32的C语言编程技术，引导读者循序渐进地掌握基于PIC32单片机的嵌入式控制系统的软硬件设计技术。全书内容分为三部分，第一部分是基础知识，第二部分是基本实践，第三部分是高级应用。

本书内容新颖实用，趣味性强，既可作为嵌入式系统设计人员的参考书，也可作为高年级本科生、研究生的学习参考书。任何对嵌入式控制系统设计感兴趣的读者都会从中受益。

图灵电子与电气工程丛书

32位单片机C语言编程：基于PIC32

◆ 著 [意]Lucio Di Jasio
译 张 鼎 岳 虹 等
责任编辑 朱 巍

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街14号
邮编 100061 电子函件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京昌平百善印刷厂印刷

◆ 开本：787×1092 1/16
印张：21.5
字数：576千字 2009年12月第1版
印数：1-3 000册 2009年12月北京第1次印刷

著作权合同登记号 图字：01-2009-5728号

ISBN 978-7-115-21612-0

定价：49.00元

读者服务热线：（010）51095186 印装质量热线：（010）67129223
反盗版热线：（010）67171154

版 权 声 明

Programming 32-bit Microcontrollers in C: Exploring the PIC32 by Lucio Di Jasio, ISBN: 978-0-7506-8709-6.

Authorized Simplified Chinese translation edition published by the Proprietor.

ISBN: 978-981-272-220-1.

Elsevier (Singapore) Pte Ltd.
3 Killiney Road
#08-01 Winsland House I
Singapore 239519
Tel：(65)6349-0200
Fax：(65)6733-1817

First Published 2009
2009 年初版

译 者 序

随着微电子技术的蓬勃发展，嵌入式控制系统正朝着微型化、功能化、智能化的方向大步前进，并已广泛应用于工业生产和日常生活中。嵌入式控制系统的核心是微处理器，而单片机则是其中使用最为广泛的一类微处理器。随着系统性能要求和任务难度的不断提高，单片机已经由经典的 8 位机发展为 16 位机以及最新的 32 位机，并且还集成了种类愈加丰富、功能愈加强大的外围设备。另一方面，由于系统功能的复杂度不断增大，嵌入式控制系统的软件设计也已由当初的汇编语言编程升级为以 C 语言为代表的高级语言编程。因此，嵌入式控制系统设计师有必要了解一些新器件，掌握一些高级语言编程技术。

本书正是在上述背景下出现的重要技术参考书，它依托最新型的 32 位单片机 PIC32 平台，详细介绍了基于 C 语言的嵌入式控制系统的软件设计方法，通过大量新颖而实用的工程实例，展示了 PIC32 单片机强大的运算处理能力和集成外围设备的丰富功能。

本书作者 Lucio Di Jasio 先生是一位经验丰富的嵌入式控制系统设计专家，曾长期从事基于 8 位单片机的系统设计工作。他结合自己从 8 位单片机升级到 32 位单片机、从汇编语言编程升级到 C 语言编程的体会，对比了 32 位单片机与 8 位单片机在运算处理能力上的区别，以及 C 语言与汇编语言在易用性方面的差异，使读者直观地感受到 32 位单片机的强大功能和 C 语言的优越性。全书在内容组织上注重循序渐进，首先介绍基础知识，使读者能够快速建立嵌入式控制系统软件的基本架构，学会基本的 I/O 操作，学会用定时器实现精确延时，掌握 PIC32 的中断系统等；然后通过精心设计的实例使读者利用 PIC32 单片机的各种片上外围设备，实现同步/异步串行通信、LCD 显示控制以及 ADC 采样等；最后，通过新颖的、趣味性极强的高级实例，使读者掌握 PS/2 键盘控制、视频显示、MMC/SD 卡接口、文件操作以及音频处理等技术。这样，既能使初学者在短时间内迅速掌握 PIC32 单片机和嵌入式控制系统 C 语言编程的关键技术，又能使经验丰富的 8 位或 16 位单片机行家掌握 PIC32 单片机的新功能，从嵌入式汇编语言设计高手轻松地转型为 C 语言编程高手。

本书主要由张鼎和岳虹翻译。Be Flying 工作室负责人肖国尊协助翻译质量和进度的控制与管理，在此予以衷心感谢。译文虽经多次修改和校正，但是由于译者的水平有限，加之时间仓促，错漏之处在所难免，我们真诚地希望同行和读者不吝赐教，译者不胜感激之至。

致　　谢

谨以此书献给我的儿子 Luca。

如果没有我妻子 Sara 超凡的支持，我是不可能完成这项工作的，她理解我，并不断鼓励我继续这项事业。我要特别感谢 Steve Bowling 和 Garry Champ，他们对嵌入式控制应用具有极高的热情与丰富的经验，都乐于无偿地审阅本书的技术内容。Garry 是初次从事这项工作，因而一开始并不清楚自己所面临的困难，但是 Steve 是我前一本书的主要技术顾问，因此他很清楚这个工作的艰辛。我还要特别感谢 Patrick Johnson，他从很早起就热情地支持本书的创意，并且排除万难使我能够直接接触到由他领导的高级设计与应用小组（该小组正在开发 PIC32 工程）。感谢“设计师”Joe Triece，他始终有求必应，并且一直都对我的工作经历饱含兴趣。感谢 Joe Drzewiecky 安装如此复杂的工具包，并且总是尽力使 MPLAB IDE 成为更好的开发平台。还要特别感谢 Nilesh Rajbharti 领导的整个 PIC32 应用小组，特别感谢 Adrian Aur、Dennis Lehman、Larry Gass 以及 Chris Smith 快速解答我提出的各种问题，并对我深入了解单片机、外围设备以及函数库的内部工作原理提供很大帮助。此外，我还要感谢我所有的朋友、Microchip 技术公司的同事以及多年来有幸一起工作的嵌入式控制工程师们。他们深深地影响了我的工作，从而造就了我在嵌入式控制领域这个神奇世界里的经历。

自从我的上一本书 *Programming 16-bit Microcontrollers in C: Learning to Fly the PIC 24*[1]出版以来，我收到了很多反馈，很多读者都写信向我表示祝贺，同时还指出错误和问题。这令我十分惭愧，但同时也使我获益匪浅，非常感谢他们！我将把读者的建议尽可能多地吸收到这本新书中，并且期望能继续得到读者的支持与建议。



① 该书中文版《16 位单片机 C 语言编程：基于 PIC 24》即将由人民邮电出版社出版，敬请关注。——编者注

引　言

几乎所有的修复程序一开始都会声明该程序存在哪些局限。因此，在本书的最开始，我也想声明：我是一个8位机程序员！

我从高中就开始用8位微控制器进行编程，并且在我职业生涯的大部分时间里，都在从事这项工作。此外，尽管我对高级语言编程也很熟悉，但是最喜欢的还是汇编语言编程！

我曾经说过，我喜欢那种能够掌握嵌入式系统在每个机器周期内的每个微秒里都做了什么的感觉。我还很沉醉于控制：我希望掌握所使用的每个外围设备的每个配置位的含义。因此，我决不轻信编译器或者其他人的函数库，除非离开它们就无法工作，或者我已经对其完全反汇编。

那么，我为何要写一本关于32位微控制器的C语言编程方面的书呢？

事实上，我是在几年前第一次接触16位微控制器后才开始了被我称为“修复程序”的工作。PIC24系列微控制器的引入使我有机会尝试并转到用C语言来对这个全新而令人激动的微控制器进行编程。而我获得的经验都写进了我的第一本书 *Programming 16-bit Microcontrollers in C: Learning to Fly the PIC24*。然而，当这本书出版的时候，Microchip公司又传出消息说32位的微控制器刚刚面世，于是我又不得不另写一本新书。

我不打算向你介绍我是如何处理第一枚测试芯片的，但是我希望你了解，我花费了很长一段时间将大多数原本为PIC24那本书设计的程序，移植到一块装有PIC32芯片的Explorer 16旧开发板上，并使这些程序都能正常运行。

Microchip公司的市场人员说，PIC32架构的微控制器是经过特别设计的，它能够将基于8位和16位PIC架构微处理器的应用程序方便而无缝地“移植”到PIC32架构上来。但是，我必须亲自检验之后才能相信它。

还有谁能比一个钟爱汇编语言、沉醉于控制的8位程序员更适合为你介绍PIC32系列芯片呢？

读者对象

PIC32是一款基于高性能32位内核处理器（MIPS）的芯片，并且包含很多支持工具、函数库和文档，因而很容易使用。本书只能使你对这个广阔的领域有个大致的了解，因此我将其称为第一次“探索”。我始终坚信学习过程应该是愉快的，并且我希望你有时间完成本书每章中那些“有趣的”练习和项目。不过，你还是需要一些必要的准备并且通过努力学习来消化所学内容，这样才能跟上本书前几章较快的进度。

本书适合于具有基本和中等水平的程序员，不适合那些“纯粹的”初学者，因为我不会讲解二进制数、十六进制数表示或者编程基础知识。尽管如此，我还是会简要地回顾一下C语言编程，因为它和最新一代通用型32位微控制器的应用有关。之后，我才会介绍更加具有挑战性的工程开发。这里，我将读者分为以下四类。

- 嵌入式控制系统的程序员：具有丰富的微控制器汇编语言编程经验，初步掌握C语言编程。
- PIC微控制器专家：初步掌握C语言编程。
- 学生或者专业人士：具备基于PC的C（C++）语言编程知识。

- 其他的 SLF（高级生命）：我知道程序员们不喜欢被简单地区分等级，因此将他们归为这一特别的类别。

尽管不同读者的技术水平和经验并不相同，但是他们都能在每章获得些感兴趣的知识。本书每章都会讲一些使你掌握 C 语言编程技巧和新型硬件设备的技术细节。如果你对这两点都很熟悉，那么就请直接跳到那一章最后的“行家”部分，或者可以考虑去做附加的练习题、阅读参考书以及访问相关网页，以便进行更深入的研究和阅读。

我之前还写过一本关于 16 位 PIC 的 C 语言编程的书，对于读过那本书的读者我还有一些特别的提示。首先我要感谢你阅读该书，接着请允许我向你解释为什么会有似曾相识的感觉。这里，我决不是用那些旧的 16 位微控制器的内容来拼凑一本新书，而是重新开发了大部分工程，以便实际展示 PIC32 架构及其工具集的关键特性：它能无缝地移植 8 位和 16 位 PIC 应用程序，它能显著提高应用系统的性能并且保持易用性。在每章的末尾，我都准备了特别的一节，以说明在程序运行过程中可能碰到的问题、如何提升性能以及其他有助于你更自信、更快速地移植应用程序的信息。

你将通过本书掌握以下内容。

- 嵌入式控制系统的 C 语言程序的结构：循环、循环、再循环。
- 基本的定时和 I/O 操作。
- 基于 PIC32 中断的多任务嵌入式控制系统的 C 语言编程基础。
- PIC32 的新外围设备：（排序不分先后）
  - 输入捕获器；
  - 输出比较器；
  - 更改通知；
  - 并行主端口；
  - 异步串行通信器；
  - 同步串行通信器；
  - 模-数转换器。
- 如何控制 LCD 显示。
- 如何产生视频信号。
- 如何产生音频信号。
- 如何访问大容量存储设备。
- 如何与 PC 共享大容量存储设备上的文件。

内容结构

本书的每一章内容都可以作为探索 32 位嵌入式编程一天的学习内容。全书包括三部分。第一部分包含篇幅较小的 6 章内容，这些内容的难度逐步增大。在每一章，我们都会研究 PIC32MX 系列微控制器的一种基本外围设备，以及使用 MPLAB C32 编译器进行 C 语言编程的一些内容。此外，在每章我们都至少会开发一个演示工程。最开始开发这种工程只需使用 MPLAB SIM 软件仿真器，而不必使用实际的硬件。不过，有时可能需要使用 Explorer 16 演示板或者 PIC32 Starter Kit。

本书第二部分是“实验”，包含 5 章。此时 Explorer 16 演示板（或者第三方的类似产品）就变得必不可少，因为有些外围设备需要实际的硬件支持才能正常测试。

本书第三部分是“扩展”，包含篇幅较大的 5 章内容。其中每章都建立在之前学习内容的

基础之上，此外还增加了新的外围设备以完成更加复杂的工程。本书第三部分开发的工程都需要使用 Explorer 16 演示板，此外还要求你具备一定的原型板设计技术（是的，你可能还得使用烙铁）。如果你没有基本的 PCB 原型开发工具，那么可以考虑使用 http://www.exploringpic32.com 网站提供的专用扩展板（AV32），它包含完成这些工程需要的所有电路和元件。

本书附带资源[①]中包含每章所开发工程的所有源代码，它们都可以直接使用。

特别说明

本书并不能替代 Microchip 公司发布的 PIC32 微控制器的数据手册、参考手册以及程序员手册，它也不能替代 MPLAB C32 编译器用户指南以及 Microchip 公司提供的函数库和相关软件工具。这些文件的电子版请去 Microchip 公司的官方网站（http://www.microchip.com）下载最新的版本。你应当熟悉这些资料并且手头常备，因为我会在书中经常引用它们，并且在需要的地方使用其中的框图或者摘录。但是请注意，本书的叙述无法替代官方资料中的信息。如果你发现书中的叙述和官方文档不一致时，请务必以后者为准，并恳请你发邮件告知我，我会非常感激你的帮助，我们会在本书相关网站（http://www.exploringpic32.com）上发布更正和提示信息。

本书也不是一本 C 语言编程的入门书。尽管本书前几章回顾了 C 语言，但是你可以从参考文献中找到该内容更好的入门课程和书籍。

检查表

尽管本书并没有像我的上一本书那样直接以航空和飞行训练举例子，但还是保留了上一本书的一些做法。

其中一个做法是在开发工程前和开发过程中使用检查表核对每一个步骤。飞行员使用检查表，不是因为使用步骤太多怕他们记不全，也不是怕他们一时忘记。事实证明，人的记忆可能会出问题，并且在处于压力的情况下更容易出错，因此他们有必要用检查表。航空飞行比其他行业更容不得出错，飞行员们把安全看得比面子重。而程序员在开发 PIC32 的代码时，误操作或者忘记某项操作并不会带来生命危险，但是，我仍然准备了很多简单的检查表，帮助你完成最常见的编程和调试任务。真心希望这些检查表无论是在你刚开始学习使用新的 PIC32 工具集时，还是在你今后像我们一样交替使用不同厂商提供的很多工程和开发环境时，都能对你有所帮助。

---

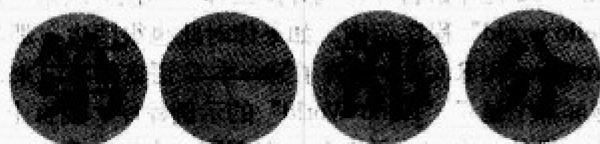
① 本书附带资源请登录图灵教育网站（www.turingbook.com）免费注册下载。——编者注

目　录

第一部分　探　索

第二部分 实 践

第三部分　扩　展



探　索

第 1 章　初识 PIC32

1.1　计划

这将是我们首次探索 32 位单片机 PIC32，有些读者可能还是首次使用 MPLAB IDE（集成开发环境）以及 MPLAB C32 程序语言开发包开发工程。即使你从未听说过 C 语言，但是你也应该听说过著名的“Hello world”程序示例。如果你对此也很陌生，那我还是说说吧。

自从几十年前 Kernighan 和 Ritchie 编著了第一本 C 语言的书以来，任何正规的 C 语言书籍都会提到一个在电脑屏幕上显示“Hello World”的示例程序。成百上千的书籍都遵从这个传统，因此本书也不例外。但是，本书的示例会略有不同，它更加真实：由于我们要设计嵌入式控制应用系统，因此我们讨论的是单片机编程。虽说所有的个人电脑或者工作站都有显示屏，但是嵌入式控制应用系统却往往并非如此。因此，在本书的第一个嵌入式应用设计中，还是采用更为基本的输出方式：数字 I/O 引脚。在后面几章介绍高级应用时，嵌入式系统将与 LCD 显示屏相接，或者通过串行端口与另一个终端相接。到那时就将实现更加高级的功能，而不只是简单地显示“Hello World”。

1.2　准备

无论你是计划一次短期的户外旅行还是筹备一次大型的北极探险，都一定要携带合适的装备。尽管对 PIC32 架构的探索决不关乎生死，但是如果你能在出门前，我的意思是在开始编写代码前，完成下列简单工作，那么你就会备感轻松。

首先，请检查一下是否安装了下列必需的软件（这些软件可以从本书附带资源获得，也可以从 Microchip 公司的 PIC32 网站 www.microchip.com/PIC32 下载最新版本）。

- ❑ MPLAB IDE，免费的集成开发环境（v8.xx 或更高版本）。
- ❑ MPLAB SIM，免费的软件仿真器（包含在 MPLAB 中）。
- ❑ MPLAB C32，C 编译器（免费的学生版）。

下面，我们将使用 New Project Setup 检查表在 MPLAB IDE 中创建一个新的工程。首先，在 Project 菜单中选择 Project Wizard。这样就会启动几个有用的对话框，在它们的指引下，只需完成几步就能有序而简洁地创建一个新工程。

（1）第一个对话框要求用户选择器件型号。请选择 PIC32MX360F512L，然后单击 Next 按钮。尽管在本例中我们只需使用仿真器，并且可以使用很多型号的 PIC32 芯片来完成本工程的任务，但是在本书的整个探索过程中都是使用的这一款芯片。

（2）在第二个对话框中，选择 PIC32 C-Complier Tool Suite，然后单击 Next 按钮。目前市面上有很多针对其他各种 PIC 架构的编译工具包，并且至少有一款能用于 PIC32 的汇编语言开发。千万不要将它们混淆在一起！

（3）在第三个对话框中，需要指定新工程的名称。也可以单击 Browse 按钮，并新建一个文件夹。将该文件夹命名为 Hello，在其中创建工程文件 Hello World，然后单击 Next 按钮。

（4）第四个对话框是向工程中添加源文件，由于这里不需要从以前的工程或者其他目录复制源文件到新工程中，因此只要单击 Next 按钮进入下一个对话框即可。

（5）单击 Finish 按钮完成创建工程。

由于这是第一次创建工程，因此还需完成以下步骤。

（6）打开新的编辑窗口。方法是选择菜单 File | New，或者按下 Ctrl+N 快捷键，或者单击 MAPLAB 标准工具条中对应的图标（New File）。

（7）输入以下 3 行注释：

```
/*
**Hello Embedded World!
*/
```

（8）选择菜单 File | Save As 将上述代码保存为文件 Hello.c。

（9）在编辑窗口上单击鼠标右键，在弹出的编辑器上下文菜单中选择 Add to Project 选项，将新建的文件加入工程中。

（10）选择菜单 Project | Save Project 保存工程。



注解 你会注意到，保存源文件后，编辑窗口里的 3 行代码会变成绿色。这是因为 MPLAB 的编辑器已经识别出该文件是 C 语言源文件（从.c 扩展名可以看出来），并且使用默认的区分上下文的染色规则。根据这些规则，注释代码显示为绿色，C 语言关键字显示为蓝色，其余代码则显示为黑色。

完成工程创建后，电脑屏幕上的工程窗口就会如图 1-1 所示。如果未看到工程窗口，那么请选择菜单 View | Project。这样 View 菜单中的选项旁就会出现小对勾。请确认已选中 File 选项卡。在后面，我们还会学习使用另一个选项卡（Symbols）。

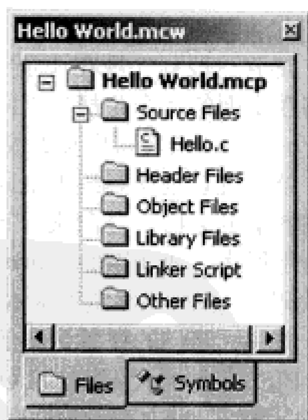


图 1-1 “Hello World”工程窗口

根据个人习惯，你可能希望将工程窗口固定在工作区某处，而不是让其处于浮动的状态。单击标题栏，从上下文菜单中选择 Dockable 选项，之后就能将工程窗口拖动到期望的屏幕边沿处，这样它就会和编辑器分开并固定下来。

1.3 探索

下面该编写代码了。我能感觉到你有些紧张，特别是你可能从未用 C 语言编写过嵌入式控制应用代码。我们要写的第一行代码是：

```
#include <p32xxxx.h>
```

这并不能算是 C 语句，而是预处理指令（用于编译器），它将在执行进一步处理前引用器件相关的文件。文件 pic32xxxx.h 中又包含更多的`#include` 指令，以便能包含与当前工程所选择的器件有关的文件。本例中将引用 p32mx360f5121.h 文件。我们也可以直接引用该文件，但是为了使代码更具独立性，并且便于将来移植到使用其他芯片的工程中，此处还是引用 p32xxxx.h。

如果你进一步查看 p32mx360f5121.h 文件的内容（这是一个普通的文本文件，可以使用 MPLAB 的文本编辑器打开它），就会发现它包含了大量定义，它们都是所选 PIC32 芯片的内部特殊功能寄存器（在很多文档里被简记作 SFR）名称的定义。如果所引用的文件准确，那么这些特殊功能寄存器的名称就和器件的数据手册以及 PIC32 参考手册中使用的名称一致。

下面是 p32mx360f5121.h 文件中的一段，包括控制看门狗模块（`WDTCON`）的特殊功能寄

存器，其中每一位都被定义了常规的名称：

```
...
extern volatile unsigned int   WDTCON__attribute__
((section("sfrs")));
typedef union {
  struct {
    unsigned WDTCLR:1;
    unsigned WDTWEN:1;
    unsigned SWDTPS0:1;
    unsigned SWDTPS1:1;
    unsigned SWDTPS2:1;
    unsigned SWDTPS3:1;
    unsigned SWDTPS4:1;
    unsigned :7;
    unsigned FRZ:1;
    unsigned ON:1;
  };
...
```

回顾源文件 Hello.c：我们要向其中添加几行，引入 main() 函数：

```
main()
{
}
```

尽管这段程序现在还是空白并且毫无用处，但已经是一个完整的 C 语言程序了。我们马上就将在上面的两个花括号之间加入嵌入式控制应用程序所需的第一部分指令。

main() 函数可以放在文件的任何位置。它无论是位于文件最顶端，还是数百万行代码之后，单片机总是在系统上电或者复位后首先执行它。这里其实做了很多简化。单片机在系统复位或者上电之后，会在执行 main() 函数之前先执行一小段由 MPLAB C32 链接器自动插入的初始化程序，即所谓的 *Startup*（启动）代码或者 *crt0* 代码（在传统的 C 语言教程中也简称为 *c0* 代码）。启动代码负责基本的内务操作，包括栈的所有重要初始化等。

首先，我们的任务是激活 PIC32 的一个或者多个输出引脚。为了和以往各代 PIC 尽可能兼容，PIC32 的输入/输出（I/O）引脚也被成组地配置在模块或者端口中，其中每一组最多包含 16 个引脚。按照字母顺序，这些模块依次被命名为 A 至 H。依照逻辑顺序，我们将首先使用 PortA。每个端口都有数个特殊功能寄存器，用于控制它们的工作。其中最重要也是最容易使用的 SFR 就是与模块同名的寄存器（比如 PORTA）。

为了区分控制寄存器名与模块名，我们将使用不同的标记：PORTA（全大写）代表控制寄存器，PortA 则代表整个外围设备模块。

根据 PIC32 的数据手册，如果将 PORTA 寄存器中的某位置 1，那么对应的输出引脚就为逻辑高电平（3.3V）。相反地，如果将某位置 0，那么对应的输出引脚就为逻辑低电平（0V）。

使用 C 语言很容易实现赋值操作，例如，我们可以在工程中增加一条赋值语句：

```
#include <p32xxxx.h>

main()
{
```

```
    PORTA = 0xff;
}
```

首先请注意，C 语言的语句必须以分号结尾。其次请注意，尽管它们很像数学方程，但它们实际上并非方程！

赋值语句首先会计算等号右侧的部分。然后，将所得结果（本例中是一个简单的十六进制常数）传入左侧的接收单元，本例中的接收单元就是单片机的特殊功能寄存器 PORTA。



注解 在 C 语言中，如果在字面值前加上前缀 0x，就表明这是一个十六进制数。以前前缀 0 被用于表示八进制数（估计现在没人使用八进制了）。其余情况下，编译器都假设是默认的十进制数。

1.4 编译与链接

既然我们已经完成了第一个 C 语言函数 main()，那么又该如何将它转换为二进制可执行文件呢？

其实，利用 MPLAB IDE 就很容易实现！只需单击鼠标执行**生成工程**（Project Build）处理即可。该处理过程十分冗长而复杂，但大致可分为两步。

（1）**编译**（compiling）。通过调用 MPLAB C32 编译器，产生目标代码文件（.o）。该文件还不是完全的可执行文件。尽管该文件中包含完整的代码，但是所有函数和变量的地址还未定义。事实上，它也可称为**可重定位的目标代码**（relocatable object code）。如果工程中包含多个源文件，那么每个文件都要执行该步骤。

（2）**链接**（linking）。调用链接器，为每个函数及变量寻找出合适的内存空间。另外，此时还能根据需要添加无限多个预编译目标代码文件以及标准库函数。链接器会产生多个文件，其中之一就是真正的二进制可执行文件（.hex）。

一旦要求 MPLAB **生成工程**，它就会很快地执行完上述步骤。工程窗口中显示的各组文件（参见图 1-1）都将参与工程生成过程的编译或链接阶段。

- ❑ Source Files（源代码文件）列表中的每个源代码文件（.c）都会被编译，并产生对应的可重定位的目标文件。
- ❑ Object Files（目标文件）列表中的每个附加的目标文件会与上一步产生的目标文件一起被链接。
- ❑ 链接阶段，链接器会使用 Library Files（库文件）列表中的文件，从中搜索并提取包含本工程所使用的函数的库模块。
- ❑ 最后，Linker Script（链接器脚本）中包含了一些含有特殊指令的其他文件，这些指令能改变每个数据段和代码段的顺序及优先级，它们会被编译到最终的二进制可执行文件中。MPLAB C32 工具包提供**默认的链接器脚本**（default linker script），它能满足大多数应用的需要，当然也能满足本书中的应用实例的要求。因此，在本书的其余地方，我们可以令工程窗口中的这部分为空，从而使用默认的配置。

工程窗口中其余两部分的处理方式有所不同。

- ❑ Header Files（头文件）部分列出了工程使用到的头文件（.h）。然而，它们并不会被编译器直接处理。将它们列在此处只是为了显示出工程的附属文件，以方便查看。如果

双击它们，就能立刻在编辑窗口中打开它们。

- Other Files（其他文件）部分包含了工程使用到的、但又不属于前面各部分的其他文件。这么做只是为了管理文档而已。

1.5　链接器脚本

和头文件 p32xxxx.h 告诉编译器器件的特殊功能寄存器的名称一样，（默认的）链接器脚本文件则告诉链接器这些特殊功能寄存器在内存中预定义的地址（这些地址由所选器件的数据手册定义）。此外，它还包括以下功能：

- 列出 FLASH 存储器的可用空间；
- 列出 RAM 存储器的可用空间；
- 列出 FLASH 以及 RAM 存储器各自的地址范围；
- 列出关键的入口地址，比如复位及异常向量的地址；
- 列出中断向量及向量表的地址；
- 列出器件配置字的地址；
- 包含附加的与处理器相关的目标文件；
- 确定软件堆和栈的地址及大小（下一章将看到，这由 MPLAB 工程文件中的参数决定）。

现在，如果你像我一样好奇，那么可能也希望了解链接器脚本文件的细节。事实上，尽管该文件的扩展名是.ld，但是它只是普通的文本文件，可以用 MPLAB 的编辑器直接打开。假如在安装 MPLAB 时选择了默认设置，就可以在下列地址找到 PIC32MX360F512L 的脚本文件 procdefs.ld：C:\Program Files\Microchip\PIC32-Tools\pic32-libs\proc\32MX360F512L。

我一定是晕了头了！要知道，我花费了半小时才在这个像迷宫一样的目录里找到该文件。而事实上，链接器会自动找到该文件并使用它，因此你根本不必为此事而担心。下面是该文件中的一段，它表述了复位向量、通用异常向量的地址以及一些其他关键入口的定义：

```
...
/*************************************************************
 *Memory Address Equates
 *************************************************************
 _RESET_ADDR        = 0xBFC00000;
 _BEV_EXCPT_ADDR    = 0xBFC00380;
 _DBG_EXCPT_ADDR    = 0xBFC00480;
 _DBG_CODE_ADDR     = 0xBFC02000;
 _GEN_EXCPT_ADDR    = _ebase_address + 0x180;
...
```

> **注解**　不要从 Windows 的资源管理器中打开 procdefs.ld 文件，也不要用 Windows 自带的记事本程序打开它，那样看着不漂亮。这是因为该文件是在 Unix 环境下创建的，它不包含 Windows 程序中使用的标准回车符。因此我建议你还是用 MPLAB 的编辑器打开它。

1.6　生成第一个工程

在 Project 菜单中选择 Build All 选项，或者单击工程工具条上的图标（Build All），

MPLAB 就会打开一个新窗口，你看到的内容应该和我看到的类似，具体参见图 1-2。

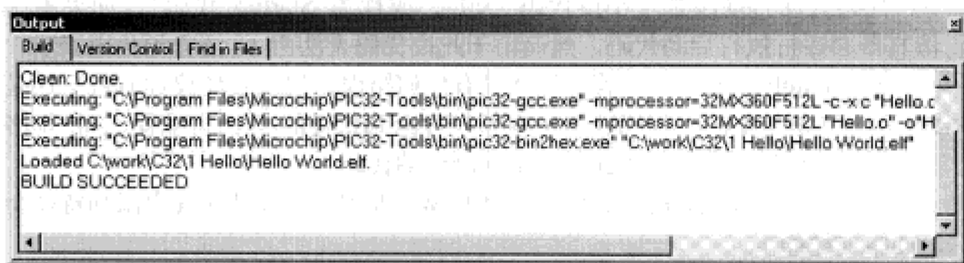


图 1-2　生成成功后，输出窗口的生成选项卡的内容

如果你更喜欢使用命令行方式，那么可以参照 MPLAB C32 编译器用户指南上的命令来调用编译器和链接器，所得的结果与 MPLAB IDE 的相同。在本书中，我们仅使用 MPLAB IDE 接口，并且利用适当的检查表使生成过程更容易。

1.7 使用仿真器

选择 Debugger | Select Tool | MPLAB SIM 选项，启动软件仿真器。我们将使用它作为本工程的调试工具。尽管这是第一次使用软件仿真器，但是我还是建议你养成使用 MPLAB SIM debugger setup 检查表配置相应参数的习惯，以便积累仿真经验。下面让我们一起完成 MPLAB 的通用配置，它们都很重要。

首先，选择 MPLAB 菜单中的 Configure | Settings 选项，随后会弹出一个巨大而复杂的对话框，请选择 Debugger 选项卡。

此处建议你选择如图 1-3 所示的 3 个选项，以便 MPLAB 自动完成以下工作。

- ❑ 在运行代码前保存编辑器窗口中所有被修改的文件。
- ❑ 在导入新的可执行文件前清除所有断点。
- ❑ 在器件复位后，将调试器的指针调整到 `main` 函数的起始处。

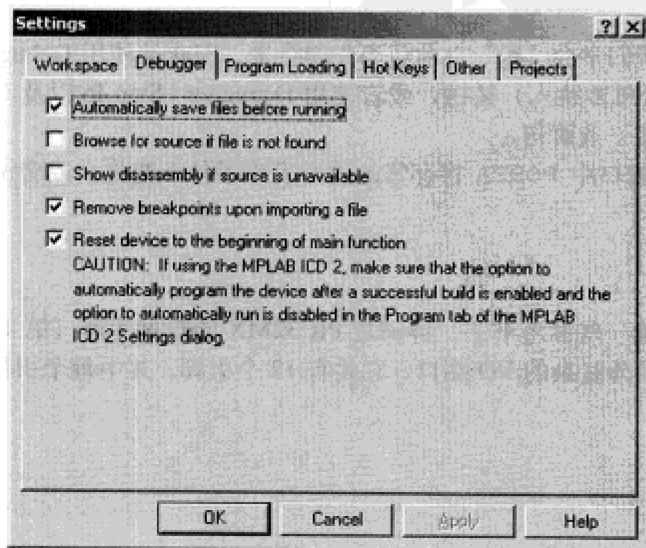


图 1-3　MPLAB 配置对话框的 Debugger 选项卡

最后一项任务看起来是多余的，但事实上并非如此。本章最开始曾提到过，链接器会在复位向量和用户代码之间自动添加一小段代码（crt0，或称为 Startup 代码）。如果不这么指示 MPLAB，仿真器会逐行执行启动代码，然而由于此段代码无法由 C 语言源代码表示，因此会弹出反汇编（disassembly）窗口。尽管这里并没有任何错误，但是我建议你有机会还是查看一下这段神秘的（并且是重要的）代码。由于本书只讨论 PIC32 的 C 语言编程，而不研究 MIPS 的汇编语言，因此，我并不准备在此详细介绍这段汇编代码。

如果一切正常，在执行代码前，我们还应打开 Watch 窗口，并且在其中添加特殊功能寄存器 PORTA，具体过程如下。

（1）从主菜单选择 View | Watch，打开 Watch 窗口（参见图 1-4）。

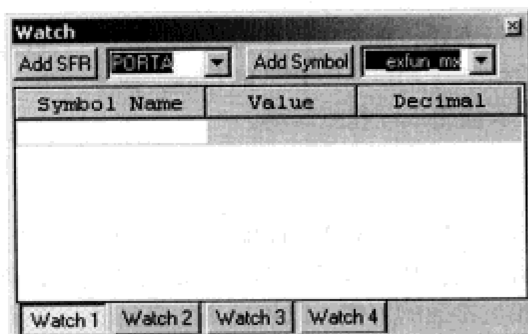


图 1-4 MPLAB IDE 的 Watch 窗口

（2）在 SFR 选择列表（位于窗口左上角）中输入或者选择 PORTA。

（3）单击 Add SFR 按钮。

（4）按下调试工具条中的仿真器复位按钮（Reset），或者选择菜单 Debugger | Reset。

（5）观察 PORTA 寄存器的内容，复位后它会被清零。

（6）同时注意在 main 函数起始处的圆括号左侧出现一个绿色的大箭头。它指示了下一步将要执行的代码。

（7）接下来，在我们学会“跑”之前应当先学会走，因此请使用 Debugger 工具条中的（单步跳出）或者（单步跳入）按钮，或者使用 Debugger | Step In 以及 Debugger | Step Over 命令，执行程序中的某一条语句。

（8）注意 Watch 窗口中 PORTA 寄存器的内容发生了什么变化。或者说，请注意是否什么也没变！令人吃惊吧！

1.8 确定方向

下面就得仔细阅读一些参考书了，特别是 PIC32MX 的数据手册（第 13 章详细介绍了 I/O 端口）。PortA 是一个极为复杂的 I/O 端口，它共有 12 个引脚，其中每个引脚都由如图 1-5 所示的逻辑电路控制。

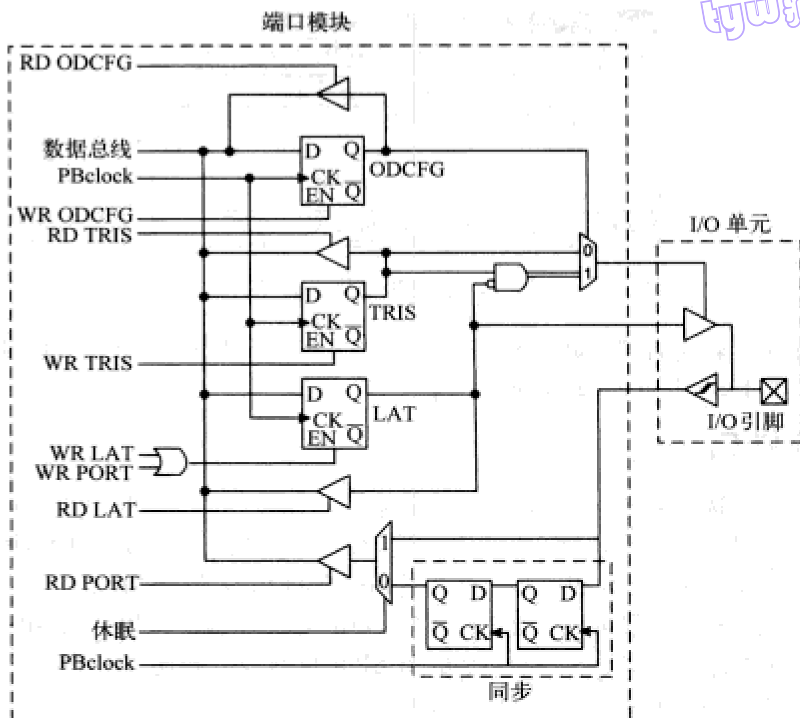


图 1-5 PIC32 的 I/O 端口的结构框图

尽管深入研究图 1-5 的框图已经超出了本章的研究范围，但还是可以简单了解一下。我们注意到，最终与引脚相连的只有 3 个信号，分别是数据输出、数据输入以及三态控制信号。其中三态控制信号决定了该引脚是作为输入还是输出，这通常被称作引脚的方向。

根据数据手册可以确定每个引脚的默认方向。事实上，在每次复位或者上电后，所有引脚的方向都将被配置为输入。这是所有 PIC 单片机都具备的标准安全特性，PIC32 也不例外。

可以使用特殊功能寄存器 TRISA 来改变 PortA 中每个引脚的方向。其功能定义很容易记忆：

- 对某位清零（0），就可将其配置为输出引脚（Output）；
- 对某位置一（1），就可将其配置为输入引脚（Input）。

因此，如果希望将 PortA 的所有引脚的方向都改为输出并且观察它们的状态变化，那么就至少还需要一个赋值语句。在增加该语句后，我们的工程就变成如下所示：

```
#include <p32xxxx.h>

main()
{
   // configure all PORTA pins as output
   TRISA = 0;
   PORTA = 0xff;
}
```

再重复执行以下步骤就可以重新测试代码。

（1）重新生成工程（既可以选择 Project | Build All，也可以按 Ctrl + F10 组合键，还可以单击工程工具栏中的 Build Add 按钮）。

（2）执行一系列的单步执行命令后，就会得到想要的结果（参见图 1-6）！

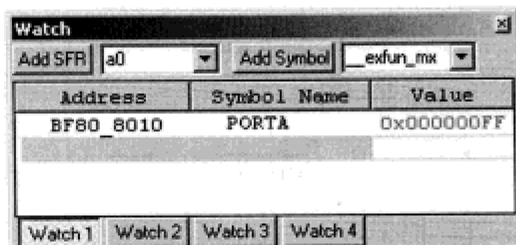


图 1-6 PortA 的内容发生变化后的 Watch 窗口

如果一切正常，就将看到 Watch 窗口中的 PORTA 的内容变成 0xFF，并以红色高亮显示。这就是我们编写的 Hello Embedded World 程序。

1.9 JTAG 端口

我们之所以首先选择 PortA，一是根据字母顺序，二是因为在 Explorer 16 演示板中，PortA 的引脚 RA0～RA7 能很方便地连接 8 个 LED。因此，如果使用在线调试器（in-circuit debugger）在实际的演示板上执行该示例代码，那么就会看到所有的 LED 都会被点亮。

此外还要注意，有一个很重要的细节会影响 PortA 部分引脚的使用。以前的 PIC 单片机采用两线协议与在线编程器或调试器（即所谓的 ICSP/ICD 接口）相连，而 PIC32 还提供了另一个在 32 位架构中广泛采用的接口——JTAG 接口。



注解 PIC24 单片机的专家们肯定会指出有些 16 位、多引脚的器件也已经提供了 JTAG 接口，以实现边界扫描（boundary scan）功能。然而在 PIC32 架构中，JTAG 的功能被进一步扩展了，还包括完整的编程与调试功能。

事实上，在实现所有的调试与编程功能时，JTAG 接口与 ICSP/ICD 接口是等效的，并且可以根据用户的个人喜好、供货条件、（Microchip 公司或者第三方提供的）开发工具的成本以及需要的引脚数量进行选择。其中，在需要的引脚数量方面，ICSP/ICD 接口比 JTAG 接口略占优势，前者所需的单片机 I/O 数量是后者的一半。然而，如果需要边界扫描功能，那么 JTAG 接口就是唯一的选择。

同时提供两个接口的结果是，PIC32 的设计师必须在器件复位或者上电时确定使用哪种调试方式。JTAG 接口的引脚与 PortA 的引脚 RA0、RA1、RA4 以及 RA5 复用，但是优先级更高。

PIC32 Starter Kit 是一套使用 JTAG 接口的编程与调试工具。MPLAB REAL ICE 以及 MPLAB ICD2 则使用传统的 ICSP/ICD 接口。

如果你打算在 Explorer 16 板上使用 MPLAB REAL ICE 或者 MPLAB ICD2 工具来测试已开发的代码，那么就要关闭 JTAG 端口，以便能够使用 PortA 的所有引脚控制 LED 灯。对应的代码如下：

```
// disable the JTAG port
   DDPCONbits.JTAGEN = 0;
```

也就是说，在 main 函数的最开始还需要增加一条赋值语句。除了要向 DDPCON 寄存器（负责配置调试数据端口）写入新值，还需使用特殊的 C 语言语句来访问某个寄存器字内的某一位（或者某些位）。在后续几章中还将详细介绍这部分内容。

如果想使用 PIC32 Starter Kit 以及一个 100 引脚的 PIM 适配器在 Explorer 16 板上测试代码，那么就绝不能关闭 JTAG 端口。但是，你仍然能够控制 PortA 上的剩余引脚：RA2、RA3、RA6 以及 RA7。不仅如此，你还可以通过 PortD 的 RD0、RD1 以及 RD2 控制 Starter Kit 板上的其他 3 个 LED。事实上，即使没有 Explorer 16 板而只有 PIC32 Starter Kit，也可以更改之前的示例程序，将所有配置 PortA 的寄存器更换为 PortD 对应的寄存器：TRISD 以及 PORTD 即可。尽管这或许没什么特别的意义，但是作为练习还是很有指导性的！

1.10 测试 PORTB

为了完成今天的学习任务，还要再研究一个 I/O 端口——PortB。控制 PortB 很简单，只需修改程序，将 PortA 的两个控制寄存器换为 TRISB 和 PORTB 即可。

重新生成工程，并完成与前面的练习相同的步骤，你就会惊奇地发现控制 PortA 的代码并不适合 PortB！

别慌，我这么做是有目的的。我希望你能体验一下 PIC32 移植中的痛苦。这将有助于你的学习并提高自身的能力。

下面要回顾一下 PIC32 的数据手册，并且更加细致地研究一下其引脚扇出图。8 位 PIC 单片机的架构与 16 位、32 位的架构有两点基本差别。

- PortB 的大部分引脚都与模-数转换器（ADC）的输入引脚复用。8 位单片机架构中保留的 PortA 引脚主要就是用作模-数转换器的输入，而在 16 位和 32 位架构中，PortA 和 PortB 的功能交换了！
- 如果某个外围设备模块的输入/输出引脚与 I/O 端口复用，那么一旦启用该外围设备，它就将完全控制该 I/O 端口，与方向控制寄存器（TRISx）的内容无关。然而，在 8 位架构中，即使该模块需要使用这些引脚，用户也得自己指定每个引脚的正确方向。

默认情况下，与“模拟”输入复用的引脚就不与“数字”输入端口相接。这就解释了本章最后的实验结果。PIC32 的 PortB 的所有引脚在上电时都被配置为模拟输入功能，因此，读取 PORTB 寄存器的结果是全零。请注意，尽管无法通过 PORTB 寄存器看到 PortB 的输出锁存内容，但其实它已经被正确配置了。查看 LATB 寄存器的内容可以验证这一点。

为了再次将 PortB 的输入引脚与数字输入端口相接，必须配置 ADC 模块。从数据手册可以知道，特殊功能寄存器 AD1PCFG 决定了每个引脚是数字型还是模拟型（参见图 1-7）。

将特殊功能寄存器 AD1PCFG 的某一位置 1，就能将该引脚从模拟型转换为数字型。新的完整程序如下：

```
#include <p32xxxx.h>

main()
{
  // configure all PORTB pins as output
  TRISB=0,                // all PORTB as output
  AD1PCFG=0xffff;         // all PORTB as digital
  PORTB=0xff;
}
```

这次，生成工程并单步运行后就会得到期望的结果（详见图 1-8）。

r-0	r-0	r-0	r-0	r-0	r-0	r-0	r-0
—	—	—	—	—	—	—	—
位 31							位 24

r-0	r-0	r-0	r-0	r-0	r-0	r-0	r-0
—	—	—	—	—	—	—	—
位23							位16

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
PCFG15	PCFG14	PCFG13	PCFG12	PCFG11	PCFG10	PCFG9	PCFG8
位15							位8

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
PCFG7	PCFG6	PCFG5	PCFG4	PCFG3	PCFG2	PCFG1	PCFG0
位 7							位 0

图例：
R=可读位　　W=可写位　　P=可编程位　　r=保留位
U=未实现位　　-n=上电复位后的取值（0, 1, x=未知）

位31~16　保留：留作将来使用，保持为0。

位15~0　PCFG<15:0>：模拟输入引脚配置控制位。
1 = 模拟输入引脚工作在数字模式，允许读取引脚的输入值，该模拟输入引脚的ADC输入复用器与AVss相连。
0 = 模拟输入引脚工作在模拟模式，无论该引脚上的电压为多少，读取数字端口的返回值总为1，引脚上的电压由ADC模块采样。

注意：AD1PCFG寄存器的功能会根据所选芯片的ADC引脚数量而变化。关于该寄存器的更多信息请参阅芯片的数据手册。

图 1-7　ADC 的引脚配置寄存器 AD1PCFG

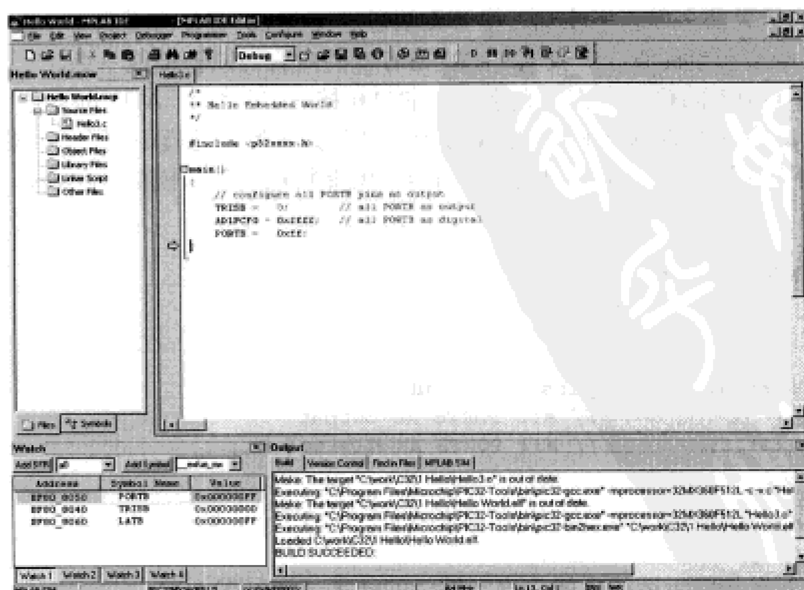


图 1-8　基于 PortB 实现的 Hello Embedded World 程序

1.11 小结

在每章结束后，都应该做简短的回顾。可以坐在一张舒适的椅子上，再来一杯冰水，好好回忆一下我们在本章所学到的东西。

编写 PIC32 单片机的 C 语言程序十分简单，至少不会比编写汇编程序或者 8 位机程序困难。根据使用端口的不同，编写两到三条不同的指令就能直接控制单片机最基本的模块 I/O 引脚与外界通信。

MPLAB C32 编译器无法读懂人类的心思。和编写汇编语言一样，需要指定正确的 I/O 端口方向。我们还要研究 PIC32 的数据手册，以便了解它与我们所熟悉的 8 位及 16 位 PIC 单片机的微小差别。

尽管正如我们想象的，嵌入式控制设备的代码与 C 语言一样是高级语言，但是我们仍然得十分熟悉所用硬件的细节。

1.12 对汇编语言行家的提示

如果你不愿无条件地承认 MPLAB C32 编译器产生的代码的正确性，那么可以在任何时候切换到反汇编列表（Disassembly Listing）视图（参见图 1-9）进行检查。由于每行 C 代码都作为注释位于它所对应的汇编代码段前面，因此可以快速查看编译器产生的代码。

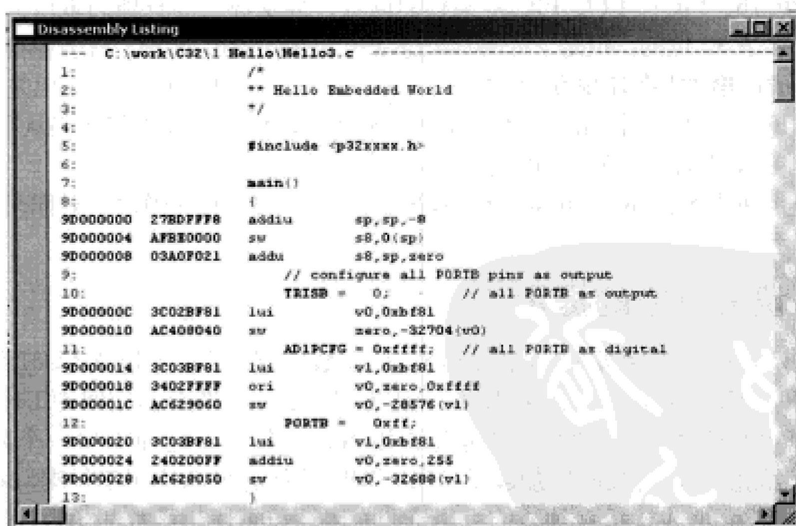


图 1-9 反汇编列表窗口

你甚至可以单步执行这些汇编代码，并在该视图下进行所有的调试工作，但是我强烈建议你不要这么做，或者只在本书前几章的练习中试试。尽管这可以满足你的好奇心，但是你要逐步学会信任编译器。使用 C 语言不但能提高你的代码生产率，还能提高代码的可读性和维护性。

再来最后一个练习，请通过选择菜单 View | Memory Usage Gauge 打开 Memory Usage Gauge（内存使用量）窗口（参见图 1-10）。

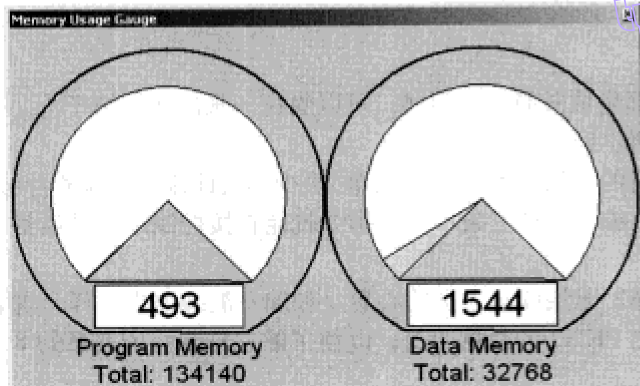


图 1-10 MPLAB IDE 的 Memory Usage Gauge 窗口

没被吓到吧，尽管我在第一个示例中仅写了 3 行代码，而它所使用的程序空间看起来已经高达 490 多字。这并不代表 C 语言的效率低下，而是因为 MPLAB C 编译器（为了我们的方便）总会自动产生一部分代码，这就是我们在前面已经提到过的启动代码（crt0）。在后文中介绍变量初始化、存储器分配以及中断时，我们还会更加详细地介绍启动代码。

1.13 对 PIC MCU 行家的提示

那些熟悉 PIC16、PIC18 以及 PIC24 架构的读者，会发现一些有趣的事情：PIC32 的所有特殊功能寄存器都是 32 位的。但是，如果你熟悉 PIC24 和 dsPIC 架构，就会惊讶地发现 PIC32 的端口数量没有成倍增加！即使现在的 PORTA 和 TRISA 是 32 位的寄存器，但是 PortA 模块仍然像 PIC24 一样只有不足 16 个引脚。在后面就会理解，这样做有利于将代码从 16 位平台移植到 32 位平台，并且对发挥 32 位架构的优势具有重要意义。

无论你之前是使用 8 位还是 16 位 PIC/dsPIC 芯片，PIC32 的外围设备对你来说都不陌生，你马上就可以上手。

1.14 对 C 语言行家的提示

我们肯定使用过标准 C 函数库里的 printf() 函数。事实上，MPLAB C32 编译器也支持该函数。但是这里是面向嵌入式控制应用，并不是为拥有数兆字节存储空间的工作站编写代码。你要习惯于控制 PIC32 单片机的低级硬件外围设备。要知道，随便调用一个库函数，比如 printf()，就可能使可执行代码增加数千字节。别以为你总可以使用串行端口和终端或者文本显示，而是要考虑到嵌入式设计领域的可用资源有限，慎重使用函数和函数库。

1.15 提示与技巧

PIC32MX 系列微处理器采用 3V CMOS 工艺，工作电压为 2.0~3.6V。因此，在大多数设备和开发板上都采用 3.3V 供电（Vdd）。尽管这会限制每个 I/O 引脚在产生逻辑高电平时的输出电压，但是它与 5V 器件和应用系统的接口十分简单。

- 为了驱动 5V 输出，请使用 ODCx 控制寄存器（PortA 对应的是 ODCA，PortB 对应的是 ODCB，等等）将每个输出引脚配置成开漏模式，并经过外部上拉电阻与 5V 电源相接。
- 数字输入引脚能承受 5V 电压，因此可以和 5V 输入信号直接相接。



注意 请注意那些还能复用成模拟输入的 I/O 引脚（比如大部分 PortB 引脚），它们最高只能承受 3.6V 的电压！

1.16 练习

如果你有一块 Explorer 16 板和一台在线调试器，那么：

- ❑ 请使用 MPLAB REAL ICE 调试或者 MPLAB ICD2 调试检查表来帮助你准备调试工程；
- ❑ 插入关闭 JTAG 端口所需的指令；
- ❑ 测试 PortA 示例程序，连接 Explorer 16 板，检查 LED0-7 的输出。

如果你有 PIC32 Starter Kit，那么：

- ❑ 请使用 PIC32 Starter Kit 调试检查表帮助你准备调试工程；
- ❑ 修改代码以便操作 PortD，但是不要关闭 JTAG 端口；
- ❑ 通过检查 PIC32 Starter Kit 上的 LED0-2 的状态来测试代码。

无论是上述哪种情况，如果能在板上找到 RB0，你就可以在 RB0 引脚上接电压表（或者数字万用表）以便测试 PortB 程序示例。当单步执行代码时，表针在 0~3.3V 之间变化表明程序正确。

1.17 参考书

Brian W. Kernighan 和 Dennis M. Ritchie 所著《C 语言程序设计》，就是程序员们常说的“K&R”或者“白皮书”。自从这本书的第 1 版于 1978 年出版以来，C 语言已经做了很多改变。第 2 版（1988 年出版）包含了更新的 C 语言的 ANSI C 标准定义，它与 MPLAB C32 编译器所使用的标准（ISO/IEC 9899:1990，也称为 C90）也更加接近。

1.18 链接

http://en.wikibooks.org/wiki/C_Programming。这是 Wiki-book 公司关于 C 语言编程的网站，它还在不断完善中。如果你不介意在线阅读，这将是很方便的资源。提示：在“A Taste of C”这一章里就能找到无处不在的“Hello World”程序示例。

第 2 章　循　　环

2.1　计划

有趣的是，很多远征的故事都是以探险者绕圈走了很久并且绝望地迷路而告终。在嵌入式控制编程中，情况恰恰相反，我们都是沿着某个圈子前进才达到目的地的：这里的程序都需要一个框架，一种能够控制代码流的结构，而这往往在*主循环*中实现。

本章首先回顾 C 语言中的基本循环语句，然后介绍第一个外围设备模块：16 位定时器 1。我们还将首次使用两个新的 MPLAB SIM 功能：*动态*（Animate）模式和*逻辑分析仪*（Logic Analyzer）视图。

2.2　准备

第 2 章所需软件与第 1 章相同（可以从本书附带资源中获得，也可以从 Microchip 公司的网站上下载最新版本），具体包括：

- ❑ MPLAB IDE（集成开发环境）；
- ❑ MPLAB SIM（软件仿真器）；
- ❑ MPLAB C32 编译器（免费的学生版）。

我们将再次使用 New Project Setup（创建新工程）检查表在 MPLAB IDE 中建立一个新工程。

请在 Project 菜单中选择 Project Wizard 选项，然后完成以下步骤。

(1) 第一个对话框要求指定器件型号。请选择 PIC32MX360F512L，然后单击 Next 按钮。

(2) 在第二个对话框中，选择 PIC32 C-Compiler Tool Suite，然后单击 Next 按钮。一定要选择 C 编译器，而不是汇编编译器！

(3) 在第三个对话框中，需要指定新工程文件的名称。也可以单击 Browse 按钮，然后创建一个新文件夹。将该新文件夹命名为 Loops，并在其中创建一个工程文件 Loops，然后单击 Next 按钮。

(4) 在第四个对话框中，由于无需从以前的工程或者文件夹中复制任何源文件，因此只需单击 Next 按钮进入下一个对话框即可。

(5) 单击 Finish 按钮完成工程向导。

(6) 选择菜单 File | New，按组合键 Ctrl + N，或者单击 MPLAB 标准工具条中的图标 🗋 (New File)，打开一个新的编辑窗口。

(7) 输入以下 3 行注释：

```
/*
** Loops
*/
```

(8) 选择菜单 File | Save As，将该文件保存为 Loops.c。

(9) 在编辑窗口中单击鼠标右键，在弹出的编辑器上下文菜单中选择 Add To Project 选项。这将告诉 MPLAB 将刚刚新建的文件加入到工程中。

（10）选择菜单 Project | Save Project 保存工程。

创建工程的步骤都是一样的，重复几次之后你就会非常熟练了，但是最好还是坚持使用本书中提到的 Create New File（创建新文件）和 Add to Project（添加到工程）这两个检查表。

2.3 探索

学完第 1 章之后，你很可能会提出这样的问题："main()函数中的代码执行完了会怎么样？"实际上什么也不会发生！

当 main()函数执行完毕并返回启动代码（crt0）时，PIC32 单片机会调用一个名为_exit()的函数以进入一个循环，直到处理器被复位才能从中跳出。请注意，这是由 MPLAB C32 编译器控制的，而并非 C 语言自带的功能。标准 C 编译器被设计成从 main()函数返回时将控制权交回操作系统，然而，正如你所见，这里并没有操作系统。



注解 与启动代码一样，_exit()函数在编辑窗口中也不可见（它不是我们编辑的代码），并且在反汇编窗口中也不可见（它也不是库函数）。查看它的唯一方法是，打开 Memory 窗口，然后选择 Code View 窗格。

幸运的是，如果你有更好的想法，也可以自己编写相应的代码以替换_exit()函数。比如，可以仿照 MPLAB C30 工具包在 PIC24 和 dsPIC 应用系统中所做的，在_exit()函数中插入一条复位指令，从而使整个应用程序反复执行。然而，我们真正希望的是嵌入式控制系统在通电期间都能连续运行。因此，让程序全部运行完毕后，复位后再重新开始运行看起来是一种简便方法，只要还有电，应用程序就能一直反复执行。

复位功能则可能用于少数条件受限的场合，但是你很快就会发现，在该"循环"中运行，就会开发出"跛行"。一旦到达程序的末尾，执行复位指令后就会使单片机返回复位向量，并再次执行启动代码。由于主程序运行时间和启动时间同样短，因此循环会非常不平衡。每次都对 SFR 和全局变量进行初始化可能并不必要，而且还会减慢应用程序的执行速度。更好的方法是，在应用程序中编写主循环代码。首先，让我们回顾一下 C 语言支持的最基本的循环控制语句。

2.4 while 循环

在 C 语言中，至少有 3 种实现循环的方法。这里介绍第一种：while 循环。其基本结构如下：

```
while ( x)
{
  // your code here...
}
```

当圆括号内的逻辑表达式 x 的返回值为真时，两个花括号之间的代码就会反复执行。然而，C 语言中的逻辑表达式又是什么呢？

首先，C 语言不区分逻辑表达式和算术表达式。它按照以下规则将布尔逻辑真（true）和假（false）表示成整数：

- 用整数 0 表示逻辑假；
- 用非零整数表示逻辑真。

因此，1 是"真"，13 和−278 也是"真"！

为了计算逻辑表达式，需要定义一些逻辑操作符，比如：

- ||，“逻辑或”操作符
- &&，“逻辑与”操作符
- !，“逻辑非”操作符

这些操作符将按照前面介绍的规则将它们的操作数视作相应的逻辑（布尔）值，并且返回一个逻辑值。以下是几个简单示例（假设 a=17 并且 b=1，即它们都为真）：

- (a || b)，真
- (a && b)，真
- (! a)，假

其次，还有一些操作符可以对数（可以是各种整型数，也可以是浮点数）比较大小，并返回逻辑值。

- ==，“等于”操作符，请注意它是由两个等号组成的，不要与前面使用过的“赋值”操作符混淆。
- ! =，“不等于”操作符。
- >，“大于”操作符。
- >=，“大于等于”操作符。
- <，“小于”操作符。
- <=，“小于等于”操作符。

以下是几个示例（假设 a = 10）。

- (a > 1)，真。
- (-a >= 0)，假。
- (a == 17)，假。
- (a ! = 3)，真。

再回到 while 循环。我们说过，一旦圆括号内的表达式产生的逻辑值为真（也就是等于任何非 0 整数），那么程序就会继续反复执行循环内代码。当表达式产生的逻辑值为假时，循环就会终止，并且会接着执行闭花括号后的第一行代码。

请注意，在执行花括号内的代码（如果有代码）前，首先要计算表达式的值。每次都是如此。

下面是一些奇特的循环程序示例：

```
while ( 0)
{
  // your code here...
}
```

上面的程序中，括号内的表达式始终为逻辑假，这意味着该循环永远不会被执行。其实这么做没什么用，事实上，它可以参加“世界上最没用的代码”竞赛！

下面是另一个示例：

```
while ( 1)
{
  // your code here...
}
```

上面的程序中，括号内的表达式始终为逻辑真，这意味着该循环会永远执行下去。这非常有用，并且事实上我们从今以后都要使用它实现主程序循环。为了增加可读性，有些人可能会考虑使用更好的方法，比如定义如下的常数：

```
#define FALSE    0
#define TRUE     !FALSE
```

然后在代码中一致地使用它们，比如：

```
While( TRUE)
{
  // your code here...
}
```

下面要向 loops.c 源文件中增加一些新代码，其中使用了 while 循环。具体的代码如下：

```
#include <p32xxxx.h>
main()
{
  // initialization
  DDPCONbits.JTAGEN = 0;    // disable the JTAG port
  TRISA =0xff00;              // PORTA pin 0..7 as output

  // application main loop
  while( 1)
  {
    PORTA = 0xff;          // turn pin 0-7 on
    PORTA = 0;               // turn all pin off
  }
}
```

这段程序的结构就是每个用 C 语言编写的嵌入式控制程序的结构。它们总是包括两部分。

- 初始化部分：包括器件外围设备的初始化以及变量初始化。这部分仅在开始时执行一次。
- 主循环部分：包含所有定义应用程序行为的控制函数。这部分要反复执行。

2.5 动态仿真

首先，请使用 Project Build（生成工程）检查表对程序 loops.c 进行编译和链接，然后使用 MPLAB SIM Simulator Setup（MPLAB SIM 仿真器配置）检查表准备好软件仿真器。

为了利用仿真器测试本例中的代码，我推荐你使用动态（animate）模式（请选择 Debugger | Animate）。在该模式下，仿真器每次只运行一行 C 程序。如果在 Watch（观察）窗口中添加了特殊功能寄存器 PORTA，就能看到它的值有节奏地交替变为 0xff 和 0x00。

动态模式下的代码执行速度可以在 Debug | Settings 对话框里配置，选择 Animation/Real Time Updates 选项卡，然后修改 Animation Step Time 参数即可，其默认值为 500ms。动态模式是一种有用且有趣的调试手段，但是它会使你对真正的程序执行时间产生错觉。实际中，如果代码需要在实际的硬件平台上执行，比如 Explorer 16 演示板（其中 PIC32 工作在 72MHz 时钟下），接在 PortA 输出引脚上的 LED 可能会闪得太快以至于人眼无法区分。事实上，每个 LED 在每秒内都要开、关数百万次。

为了将 LED 的闪烁速度降低到每秒几次，我建议使用定时器来实现。这样我们还可以从中学习使用 PIC 单片机的重要片上集成外围设备。本例中将使用定时器 1，它是 PIC32MX360FJ512L 芯片（参见图 2-1）的 5 个片上定时器之一，也是最灵活、最简单的外围设备模块之一。只需快速浏览一下 PIC32 的数据手册，查看定时器 1 的框图及其控制寄存器的详细信息，就能得到理想的配置参数。

定时器1共有3个特殊功能寄存器：

- ❑ TMR1，用于存放16位计数值；
- ❑ T1CON，控制定时器使能，配置定时器的工作模式；
- ❑ PR1，用于产生周期性的定时器复位信号（本例中并不需要）。

首先对 TMR1 清零，以便从零开始计数，对应的代码为：

```
TMR1 = 0;
```

图 2-1　16位定时器1模块的框图

然后初始化 T1CON（见图 2-2），以便定时器能够工作在简单配置模式。

- ❑ 激活定时器1：TON = 1。
- ❑ 使用单片机的主时钟作为计数时钟源（Fpb）：TCS = 0。
- ❑ 预分频系数取最大值（1:256）：TCKPS = 11。
- ❑ 由于采用单片机的内部时钟作为定时器时钟，因此不必使用输入门驱动以及同步功能：TGATE = 0，TSYNC = 0。
- ❑ 不关心定时器在 IDLE 模式下的行为：SIDL = 0（默认值）。

虚拟地址	名称		位 31/23/15/7	位 32/22/14/6	位 29/21/13/5	位 28/20/12/4	位 27/19/11/3	位 28/18/10/2	位 25/17/9/1	位 24/16/8/0
BF80_0600	T1CON	31:24	—	—	—	—	—	—	—	—
		23.16	—	—	—	—	—	—	—	—
		15:8	ON	FRZ	SIDL	TMWDIS	TMWIP	—	—	—
		7:0	TGATE	—	TCKPS<1:0>		—	TSYNC	TCS	—

图 2-2　T1CON：定时器1的控制寄存器

如果将上述配置位组合在32位的数据内，同时向 T1CON 赋值，则有：

```
T1CON = 1000 0000 0011 0000
```

或者采用更简单的十六进制表示，则为：

```
T1CON = 0x8030;
```

一旦完成对定时器的初始化，就可以进入循环等待 TMR1 到达由常数 DELAY 指定的数值。对应的代码为：

```
while( TMR1 < DELAY)
{
  // wait
}
```

假设所用的外围设备总线时钟频率为 36MHz，那么为了实现 1/4 s 延时，就需要给 DELAY 指定一个非常大的数值。循环产生的总延时可由以下公式计算得出：

$$Tdelay = (Fpb)\times 256 \times DELAY$$

如果 Tdelay=256ms，那么 DELAY 就等于 36000，因此：

```
#define  DELAY  36000
```

在主循环中的每条 PORTA 赋值语句前放一段这样的延时循环程序，就得到了最终程序：

```
/*
** Loops
*/
#include <p32xxxx.h>

#define DELAY 36000          // 256ms

main()
{
  // 0. initialization
  DDPCONbits.JTAGEN = 0;  // disable JTAGport, free up PORTA
  TRISA = 0xff00;         // all PORTA as output
  T1CON = 0x8030;         // TMR1 on, prescale 1:256 PB=36MHz
  PR1 = 0xFFFF;           // set period register to max

  // 1. main loop
  while( 1)
  {
    //1.1 turn all LED ON
    PORTA = 0xff;
    TMR1 = 0;
    while ( TMR1 < DELAY)
    {
      // just wait here
    }

    // 1.2 turn all LED OFF
    PORTA = 0;
    TMR1 = 0;
    while ( TMR1 < DELAY)
    {
      // just wait here
    }
  } // main loop
} // main
```



注解　在C语言编程中，随着代码长度的增加，花括号的数量也急剧增加。即使坚持最好的缩进原则，只需一小会儿，也可能很难记得每个闭花括号对应哪个开花括号。在闭括号旁添加一点儿提示（注释），就能使得代码的可读性更好。此外，在编辑窗口中使用快捷键Ctrl+M，还能在代码中匹配的花括号之间快速跳转。

下面将生成工程，并验证它能否正常工作。如果你有Explorer 16演示板，就可以立即运行该程序。板上的LED灯将以舒适的慢节拍闪烁，频率大约为每秒两次。

尽管利用MPLAB SIM仿真器也能运行这段程序，但是运行速度太慢了。我不知道你的PC有多快，在我的电脑上，MPLAB SIM无法达到接近实际PIC32单片机的运行速度。

如果使用动态模式，情况会更糟糕。正如我们前面看到的，动态模式会在执行每行代码之间都增加大约半秒延时。因此，在纯粹为了调试代码时，可以将DELAY常数调整为更小的值，比如36。

2.6　使用逻辑分析仪

为了完成本节课的内容，并且使其更加有趣，在生成工程后，我建议尝试一种新的仿真工具：MPLAB SIM逻辑分析仪。

该逻辑分析仪能记录器件的多个输出引脚上的值，并提供高效的图形化视图，但是要特别注意对它的初始化。

首先，要确定仿真器的跟踪（Tracking）功能已经打开。

（1）选择Debug | Settings对话框，然后选择Osc/Trace选项卡。

（2）在跟踪选项部分，请选择Trace All选择框。

（3）通过菜单View | Simulator Logic Analyzer打开Analyzer窗口（见图2-3）。

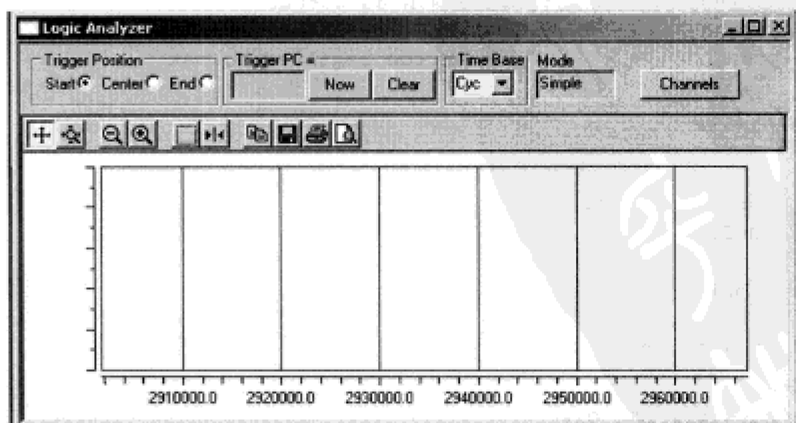


图2-3　MPLAB SIM逻辑分析仪窗口

（4）单击Channels按钮，弹出通道选择对话框（见图2-4）。

（5）在这里，可以选择希望可视化观察的器件输出引脚。本例中，请选择RA0，然后单击Add=>按钮。

（6）单击OK按钮，关闭通道选择对话框。

为了便于将来参考，请将前面所有的步骤列在逻辑分析仪配置检查表中。

(7) 单击调试器工具条上的 ▷ (Run) 按钮，或者选择菜单 Debugger | Run，或者按下快捷键 F9，都可以启动仿真。

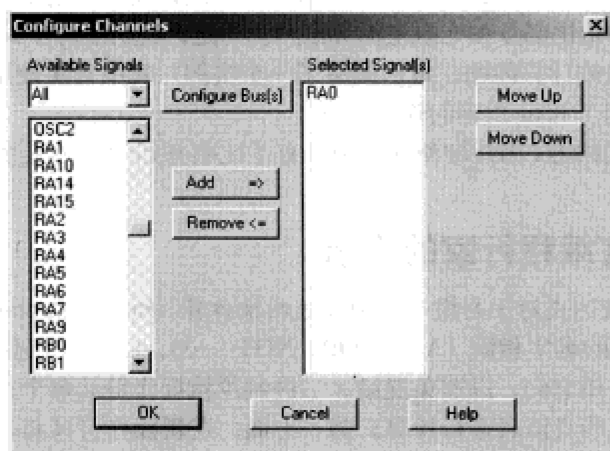


图 2-4　逻辑分析仪通道配置对话框

(8) 过一会儿，请按下调试器工具条上的 ⏸ (Halt) 按钮，或者选择菜单 Debugger | Run，或者按下快捷键 F5，停止仿真。

逻辑分析仪窗口会显示一个整齐的方波图，具体见图 2-5。

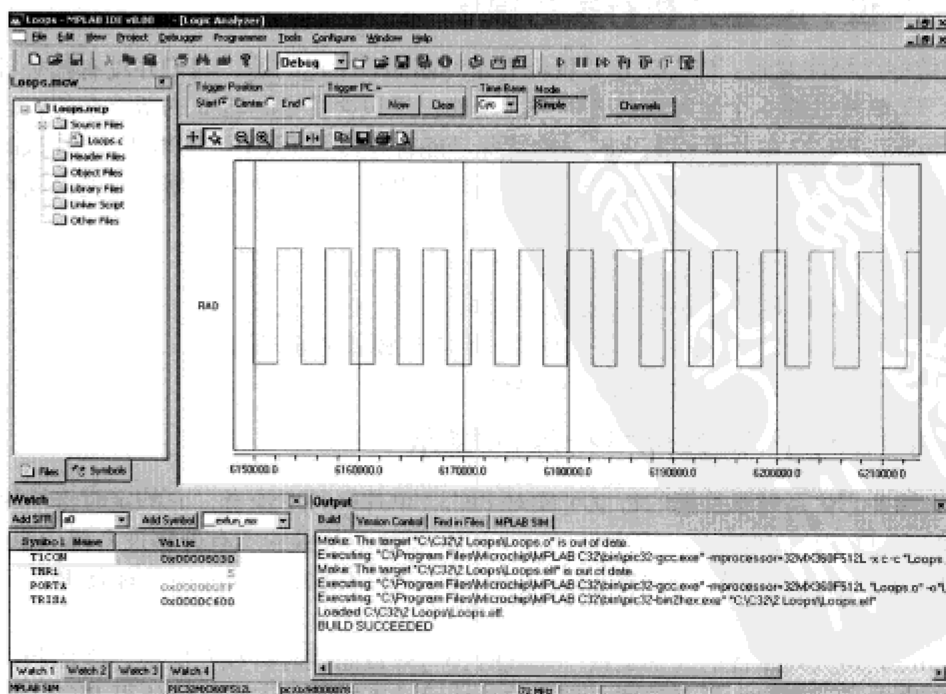


图 2-5　Loops 工程运行后的逻辑分析仪窗口

2.7 小结

本章研究了 MPLAB C32 编译器处理程序结束的方法，还首次给工程增加了一点儿结构，将 `main()` 函数分为初始化部分和无限主循环部分。为此，我们学习了 `while` 循环语句，并且初步接触了逻辑表达式的计算问题。最后例举了一个实例，其中首次使用了定时器模块，并且利用逻辑分析窗口绘制了 `RA0` 引脚的输出。

后文还将涉及上述内容，因此即使你此刻拥有了比刚开始学习时更多的疑问，也不必担心；学习的过程就是如此。

2.8 对汇编语言编程行家的提示

C 语言中的逻辑表达式对于习惯于使用二进制操作符（binary operators）的汇编语言程序员来说有些诡异，它们的名字相同（AND，OR，NOT……）。C 语言中还有很多**二进制操作符**，但是我故意在本节不使用它们，以避免混淆。二进制逻辑操作符从每个操作数中取出相对的一组数据位，并根据定义的真值表进行计算。另一方面，逻辑操作符又将每个操作数（与所包含的位个数无关）看作单独的布尔值。

看看下面对单字节操作数的计算示例：

```
                11110101                      11110101       (真)
二进制或        00001000        逻辑或        00001000       (真)
                --------                      --------
    得          11111101          得          00000001       (真)
```

2.9 对 8 位 PIC 单片机行家的提示

我相信你已经注意到：PIC32 单片机没有定时器 0！幸运的是，我们并没有损失什么。事实上，PIC32 单片机的 5 个定时器具备的强大功能已经包含了定时器 0 的所有功能，因此你根本不必怀念它。PIC32 单片机控制定时器的特殊功能寄存器与 PIC16 和 PIC18 单片机的名称类似，并且在结构上保持高度一致。此外，只要看一眼 PIC32 单片机的数据手册就会发现，设计师还设法增加了一些新特性，包括：

- 所有的定时器都是 16 位宽；
- 每个定时器都有一个 16 位的周期寄存器；
- 定时器 2/3 以及定时器 4/5 可以组成新的 32 位模式的定时器；
- 定时器 1 增加了新的外部时钟门控特性。

2.10 对 16 位 PIC 单片机行家的提示

PIC24 和 dsPIC 系列单片机的行家可能对 PIC32 单片机并不惊奇，它的定时器模块被设计成与之前的 16 位架构高度兼容。事实上，PIC32MX 系列单片机的所有外围设备模块都是如此，并且和 PIC24H 系列非常相似。此外，到处都升级到 32 位的总线还时不时地能使 PIC32 的性能明显增强。

尽管如此，最具戏剧性的区别是核心总线时钟与外设总线时钟相互解耦。这是 PIC 单片机的架构在历史上第一次彻底与之前所有型号的总线设计不同的地方。这就使得 PIC32 单片机的 MIPS 内核脱离了 Flash 存储器阵列以及外围设备模块的速度限制，从而能在实现更高性能的同时又不影响兼容性，并且运行功耗极低。下一章还将详细研究这两种内部总线、振荡器模块以及它们的配置。

2.11 对 C 语言行家的提示

如果你习惯于在个人电脑或者工作站上进行 C 语言编程，那么可能会期望 main() 函数结束时能将控制权交回给操作系统。尽管也有一些实时操作系统（real-time operating system, RTOS）支持 PIC32 单片机，但是很多应用不需要也不能使用它们，本书中的简单示例就是如此。默认情况下，MPLAB C32 编译器认为不必将控制权返还给操作系统。

2.12 对 MIPS 行家的提示

读者当中的 MIPS 行家可能已经在寻找前面提到的核心 32 位定时器（是的，PIC32 实际上共包含 6 个定时器）以及可通过协处理器 0（coprocessor 0，CP0）指令访问的硬件控制寄存器。它们的确很诱人，但是在这里我故意避开了，并且尽可能不使用它们。由于 MIPS 内核已经嵌入到 PIC 芯片内部，因此你只需要熟悉 PIC 单片机的外部特性即可。通过这里的介绍可以发现，尽管 PIC32 是迄今设计的最快的 PIC 单片机，但是其内核及外围设备的用法与普通 PIC 单片机一样，它仍然是纯正的 PIC 单片机。

2.13 提示与技巧

有些嵌入式设备需要连续运行数月或者数年，而且器件从不关断也不会收到复位命令。但是单片机的控制寄存器只是普通的 RAM 存储器单元。电源波动（无法被掉电复位电路检测到）、附近的噪声设备发出的电磁脉冲，甚至是宇宙射线，都有可能使存储器单元的内容发生微小变化。如果运行时间足够长（很多年），加上应用的特殊性，的确可能发生这种情况。当设计的系统需要可靠地长时间工作时，就必须仔细考虑周期性地“刷新”应用系统所用的关键外围设备的大部分关键控制寄存器的内容。

初始化命令要有序地分组成一个或多个函数。一旦上电，就要首先调用这些函数，然后才能进入主循环。此外，还要确保当处理器空闲并且没有其他紧急任务等待处理时，在主循环内部能够调用这些初始化函数，以便周期性地重新初始化每个控制寄存器。

2.14 使用外围设备函数库的提示

MPLAB C32 工具包包含一个标准 C 函数库的完整集，以及一个专门设计用于简化和标准化 PIC32 单片机所有内部资源的外围设备函数库（peripherals libraries）的附加集。外围设备函数库专门设计成与之前的 Microchip 16 位架构、特别是 PIC24 系列单片机高度兼容。下面的示例通过使用定时器的函数库 timer.h 来演示使用函数库的优缺点。

既然要用外围设备函数库来初始化定时器 1，正如今天开发的 loops 工程进行的初始化一样，那么就要替换直接访问定时器寄存器部分的代码：

```
TMR1 = 0;
T1CON = 0x8030; // or TMR1bits.ON = 1; TMR1bits.TCKPS=3;
PR1 = 0xFFFF;
```

这里将改用下述代码：

```
WriteTimer1( 0);
OpenTimer1( T1_ON | T1_PS_1_256, 0xFFFF);
```

使用函数库的明显优点是无需为上面的两行代码增加注释，它们已经相当清晰了。这种代码本身就可以作为说明文档。此外，如果拼错了其中某个参数名，编译器会迅速报错并指出错

误的位置。

当然，它也并非十全十美。尽管编译器可以检查出函数的参数的拼写错误，但是在大多数情况下，编译器无法指出用户是否对正确的函数使用了正确的参数。比如，当配置定时器2时，下面的错误就无法被检查出来：

```
OpenTimer2( T2_ON | T1_PS_1_256, 0xFFFF);
```

看起来这是个十分幼稚的错误，但是这可能导致你花费数个小时来冥思苦想：既然定时器2的预分频器配置是由编译器完成的，为何还会出错？

使用函数库的最大优点是，它们具有很好的抽象性，但同时它又是一个潜在的故障源。由于它们向设计者隐藏了实现细节，因此无法得知，比如TMR1寄存器是会被函数OpenTimer1()清零呢，还是要求在调用该函数前由用户对它们清零。即使看起来该函数并未清零TMR1，但是必须进入函数源文件或者在反汇编列表中检查才能验证。

此外，尽管PIC32MX器件的数据手册定义了所有控制寄存器（比如T1CON）及每一位（比如TCKPS）的官方名称，但是函数库中的参数名称和拼写还有所不同（比如T1_PS_1_256）。尽管设计师已经尽力让二者的名称近似，但是仍有区别。这些新名称的定义只能在文件的特定部分找到。因此你必须查阅 *Peripheral Library User Guide*（外围设备函数库用户指南），或者查看头文件timer.h，以便确认每个参数的定义。

因此，我个人建议，必须针对具体情况并经过谨慎的考虑后才能使用外围设备函数库。对于像I/O端口以及定时器这种简单的外围设备来说，我看不出使用函数库有何优势。毕竟，只有通过研究每个控制寄存器的每一位，并且熟悉它们的含义及相互关系，才能选择出正确的参数。除此之外，就剩下WriteTimer1(0);语句，难道它真的比TMR1=0的可读性好得多吗？

当外围设备模块越来越复杂，而使用库函数又会给我们带来方便时，我建议最好还是使用它们。例如，本书后面就将使用DMA函数库。

然而，在本书其他地方，你会看到大多数示例会同时使用这两种方法。无论如何，这都是由个人的编程风格决定的，你觉得使用哪个更方便就用哪个，同时用两个也行。

2.15　练习

（1）在PortA输出一个计数器，而不是改变它们的开关状态。如果采用PIC32 Starter Kit，请使用PortD。

（2）输出成跑灯的样式，而不是闪烁的样式。

（3）使用专用的外围设备库函数来控制PortA的引脚，重写loops工程；启动定时器、配置定时器并且读取计数值；如有必要，请关闭JTAG端口。

2.16　参考书

Larry Ullman和Marc Liyanage所著 *C Programming* 。这本书会一步一步地指导你学会C语言编程。

2.17　链接

http://en.wikipedia.org/wiki/Control_flow#Loops。以广阔的视角介绍编程语言以及与编码和控制循环相关的问题。

http://en.wikipedia.org/wiki/Spaghetti_code。当你编写的循环陷入死循环时，程序就可能失控。

第 3 章　循环和数组

3.1　计划

第 2 章介绍了每个嵌入式控制应用软件的核心都有一个循环，以及如何使用 C 语言的 while 语句实现这个循环。在本章中，我们将继续探索 C 语言程序员实现循环的其他方法。顺着这个思路，我们将简要回顾整型变量声明及递增和递减运算符，并快速了解数组的声明与使用。在本章的最后，我们将创建一个很有趣的工程，它会用到本章学到的所有知识。这个工程可以看作创建一个求生工具，倘若你搁浅在一个荒芜的岛屿上，这个工具就会非常有用。

3.2　准备

这里将继续使用 MPLAB SIM 软件仿真器，除此之外还要增加一块 Explorer 16 演示板。在准备新的演示项目时，你可以用 New Project Setup（创建新工程）检查表来创建一个名为 Message 的新工程和一个名为 Message.c 的源文件。

3.3　探索

在 while 循环中，仅当逻辑表达式的返回值为布尔真（非零）时，花括号之间的程序块才会被执行。其中，逻辑表达式是在执行循环之前计算的，也就是说如果表达式的返回值为假，那么循环体内的程序就一次也不会被执行。

3.4　do 循环

如果循环至少要被执行一次，然后再根据逻辑表达式的值决定是否继续反复执行，那么就请看看下面的其他循环结构。

首先要介绍的是 do 循环语句：

```
do {
  // your code here...
} while ( x);
```

尽管上述 do 循环也使用了 while 关键字，但是千万不要被此迷惑，它们的行为完全不同。

在 do 循环中，总是先执行一遍花括号内的代码，然后再计算逻辑表达式的值。当然，如果是为 main() 函数设计一个无限循环，那么使用 do 或者 while 是没有区别的：

```
main()
{
  // initialization code
  ...
  // main application loop
  do {
    ...
  } while ( 1)
} // main
```

请看下面的特殊情况，我们将分析该循环的执行过程：

```
do{
  // your code segment here...
} while ( 0);
```

你会发现，循环体内的代码仅会被执行一次，即循环体代码周围的代码都是无用的，这个程序也可以参加“世上最没用的代码”大赛了。

下面再看一个更有用的示例，它使用 while 循环使某段代码重复执行预定的次数。首先需要一个变量作为计数器。换句话说，还需要一个 RAM 存储器单元保存计数器的数值。



> **注解**　在前两章中，由于我们仅使用到预定义变量（PIC32 的特殊功能寄存器），因此几乎完全跳过了变量声明部分。

3.5　变量声明

整型变量的定义如下：

```
int i;
```

由于使用关键字 int 将 i 声明为 32 位（有符号）整数，因此 MPLAB C32 编译器就会为此安排 4B 存储空间。后面，链接器将决定这 4B 在所用 PIC32 器件的物理 RAM 存储器中的位置。这里的变量 i 可以从最小负值-2 147 483 648 计数到最大正值+2 147 483 647。这个范围很大，甚至和大多数 8 位及 16 位编译器的更高一级的整数类型 long 的范围一样大。长整型的定义如下：

```
long l;
```

但这正是 32 位单片机的优势。PIC32 的 ALU（arithmetic and logic unit，算术逻辑单元）处理 32 位整数和处理 16 位或者 8 位整数的开销是一样的（所需的时钟周期一样）。因此，MPLAB C32 编译器默认地将 32 位作为基本整型（int），而 long 则是 int 的同义词。

从性能角度来看这是很好的，但是它是以消耗内存空间为代价的。PIC32 的 RAM 存储器存储每个整型变量所花费的空间是 8 位或者 16 位 PIC 单片机的两倍。尽管 PIC32 芯片的存储器空间更大，但是它的 RAM 空间常常是嵌入式控制应用中最珍贵的资源之一。

因此，如果不必使用 PIC32 的 int 和 long 型整数所表示的大范围数据，而想找个较小的计数器，并且所需要的范围不超过-128~+127，那么就可以改用 char 整型，其定义如下：

```
char c;
```

MPLAB C32 编译器将使用 8 位空间（单字节）来保存变量 c。

如果需要的数据表示范围为-32768~+32767，那么 short 整型最适合，其定义为：

```
short s;
```

MPLAB C32 编译器将使用 16 位空间（双字节）来保存变量 s。上述 4 种整数类型都可以添加 unsigned 属性，即：

```
unsigned char  c;          // ranges from 0..255
unsigned short  s;         // ranges from 0..65,535
unsigned int    i;         // ranges from 0..4,294,967,295
unsigned long   l;         // ranges from 0..4,294,967,295
```

如果你确实需要使用大范围数据，那再没有比 long long 类型以及带有 unsigned 属性

的 long long 类型更大的了：

```
long long  l;                // ranges from -2^63 to+ 2^63-1
unsigned long long  l;       // ranges from 0 to + 2^64
```



> **注解** MPLAB C32 编译器将为每个 long long 类型的变量分配 64 位空间（相当于 8B）。这个空间看起来很大，但是 PIC32 使用它们和 PIC16 使用 16 位整数的工作量差不多。

下面介绍可用于浮点计算的变量类型定义：

```
float        f;              // defines a 32 bit floating point
long double  d;              // defines a 64 bit floating point
```

但是，针对我们所需的循环体设计来说，目前只需注意整型即可。

3.6 for 循环

再回到计数器示例，我们只需使用简单的整型变量作为序号/计数器，所需的计数范围是 0～5。因此，使用 char 整型即可：

```
char i;                     //declare i as an 8-bit integer with sign

i = 0;                      // init the index/counter
while ( i<5)
{
  // insert your code here ...
  // it will be executed for i= 0, 1, 2, 3, 4

  i = i+1;                  // increment
}
```

在日常编程中，我们会遇到许多递增或递减计数任务。在 C 语言中，还有第三种为此专门设计的循环语句，这就是 for 循环，它能方便地实现递增或递减计数任务。下面就是将 for 循环用于前面示例的代码：

```
for ( i=0; i<5; i=i+1)
{
  // insert your code here ...
  // it will be executed for i=0, 1, 2, 3, 4
}
```

你一定也认为 for 循环语句十分简单，并且更容易书写。后面还会看到它还更便于阅读和调试。for 关键字后面的圆括号内被分号隔开了 3 个表达式，它们正是在前面的示例中使用的 3 个表达式：

- ❑ 初始化序号；
- ❑ 通过逻辑表达式检查是否到达计数终点；
- ❑ 改变序号/计数器值，这里是递增变化。

可以将 for 循环视作 while 循环的简化形式。事实上，它也要先计算逻辑表达式的值，如果第一次就是逻辑假，那么循环体圆括号内的代码就一次也不会执行。

借此机会，还要回顾 C 语言的另一个方便的缩写形式。C 语言有一组用作递增和递减运算的特殊保留符号：

++　递增，如 i++，等价于 i = i+1；

--　递减，如 i--，等价于 i = i-1；

就介绍这么多了，后面几章还会详细讨论它们。

3.7 更多循环示例

下面介绍更多关于 for 循环的示例，其中都使用了递增/递减运算符。第一个示例是从 0 数到 4：

```
for ( i=0; i<5; i++)
{
  // insert your code here ...
  // it will be executed for i= 0, 1, 2, 3, 4
}
```

第二个示例是从 4 数到 0：

```
for ( i=4; i>=0; i--)
{
  // insert your code here ...
  // it will be executed for i= 4, 3, 2, 1, 0
}
```

能不能用 for 循环实现一个（无限的）主程序循环呢？当然可以！下面就是一个示例：

```
main()
{
  // 0. initialization code
  // insert your initialization code here...

  // 1. the main application loop
  for ( ; 1; )
  {
    // insert your main loop here...
  }
} // main
```

如果你愿意，就可以使用上述形式的主循环。对我而言，从现在开始则只使用 while 语句实现它（这只是个习惯而已）。

3.8 数组

在开始编写下一个工程的代码前，还要再回顾一个 C 语言的概念：数组变量类型（array variable type）。数组是一种包含一定数量相同类型数据的连续存储区。一旦定义了数组，就可以通过数组名称和序号来访问数组中的每个数据。数组的声明也和单个变量的声明一样简单，只要在变量名后的方括号内添入数据个数即可，例如：

```
char c[10];      // declares c as an array of 10 x 8-bit integers
short s[10];     // declares s as an array of 10 x 16-bit integers
int i[10];       // declares i as an array of 10 x 32-bit integers
```

方括号还用于表示数组中的元素或者为数组元素赋值，如：

```
a = c[0];            // copy the value of the 1st element of c into a
c[1] = 123;          // assign the value 123 to the second element of c
i[2] = 12345;        // assign the value 12,345 to the third element of i
i[3] = 123* i[4];    // compute 123 x the value of the fifth element of i
```





注解 在C语言中，数组的第一个元素的序号是0，而最后一个元素的序号是 (N-1)，其中 N 是声明的数组长度。

使用 for 循环来操作数组是非常方便的。下面将通过示例来说明这一点。声明一个长度为10的整数型数组，并希望将数组的每个单元都初始化为1，相关的代码如下：

```
int a[10];             // declare array of 10 integers: a[0], a[1],
                          a[2] ... a[9]
int i;                 // to be used as the loop index
for ( i=0; i<10; i++)
{
  a[ i] = 1;
}
```

3.9 发送一条信息

下面，我们要将目前已经回顾的所有C语言知识应用于下一个工程。这次，我们要利用连接在 PortA 上的整排 LED 灯与外界通信。由于 Explorer 16 演示板上恰好连着这些 LED，因此可以让它们快速有序地闪烁，通过左右有节奏的移动来显示一条短文本信息。

显示“Hello World!”或者只显示“HELLO”的代码如下：

```
#include <p32xxxx.h>

// 1. define timing constants
#define SHORT_DELAY      400
#define LONG_DELAY       3200

// 2. declare and initialize an array with the message bitmap
char bitmap[30] = {
  0xff,     // H
  0x08,
  0x08,
  0xff,
  0,
  0,
  0xff,     // E
  0x89,
  0x89,
  0x81,
  0,
  0,
  0xff,     // L
  0x80,
  0x80,
  0x80,
  0,
  0,
  0xff,     // L
  0x80,
  0x80,
  0x80,
  0,
```

```
    0,
    0x7e,     // O
    0x81,
    0x81,
    0x7e,
    0,
    0
    };

// 3. the main program
main()
{
  // disable JTAG port
  DDPCONbits.JTAGEN = 0;

  // 3.1 variable declarations
  int i;                      // i will serve as the index
  // 3.2 initialization
  TRISA = 0xff00;        // PORTA pins connected to LEDs are outputs
  T1CON = 0x8030;        // TMR1 on, prescale 1:256 Tpb=36MHz
  PR1 = 0xFFFF;          // max period (not used)

  // 3.3 the main loop
  while( 1)
  {
    // 3.3.1 display loop, hand moving to the right
    for( i=0; i<30; i++)
    { // update the LEDs
      PORTA = bitmap[i];
      // short pause
      TMR1 = 0;
      while ( TMR1 < SHORT_DELAY)
      {
      }
    } // for i

    // 3.3.2 long pause, hand moving back to the left
    PORTA = 0;                // turn LEDs off
    // long pause
    TMR1 =  0;
    while ( TMR1 < LONG_DELAY)
    {
    }
  } // main loop
} // main
```

在第一部分中，我们定义了一些时间常数，以便控制运行和调试时显示序列的移动速度。

在第二部分中，我们声明并初始化了一个含有 30 个元素的 8 位整数数组，其中每个元素都包含显示队列中的一个 LED 配置。



提示 请在一张白纸上将数组初始化时使用的十六进制数转换为二进制数，然后用记号笔或者红色的笔把每个 1 标记出来，就可以看出其中的信息了。

第三部分是主程序，首先是变量声明部分（3.1），接着是单片机初始化部分（3.2），最后

是主循环（3.3）。

主循环（while 循环）又可以进一步分成两部分：3.3.1 包括实际的 LED 闪烁序列，共有 30 步，在板上从左向右扫描显示。其中 for 循环用于依次读取数组中的每个元素，而 while 循环则用于等待定时器 1 到达合适的移动间隔时间。3.3.2 包含一个扫描停顿，它由 while 循环等待更长的 Timer1 延时实现。

3.10 用逻辑分析仪进行测试

为了测试上面的程序，我们将先用 MPLAB SIM 软件仿真器和逻辑分析仪窗口进行验证，具体步骤如下。

（1）利用 Project Build（生成工程）检查表对工程进行编译。

（2）打开 Logic Analyzer（逻辑分析仪）窗口。

（3）单击 Channel 按钮，依次添加与 LED 灯相连的 I/O 端口 RA0～RA7。

MPLAB SIM 配置与 Logic Analyzer Setup（配置逻辑分析仪）检查表能保证你不出错。

（4）返回编辑器窗口，并将光标移到 3.3.2 节的第一行代码处。

（5）选择 context（上下文）菜单中的 Run to Cursor（运行到光标处）命令。这样程序就会执行完整个消息输出部分（3.3.1），并且在进入长延时前停止。

（6）一旦仿真停止在光标所在行，就可以切换到 Logic Analyzer（逻辑分析仪）窗口并检查输出波形。如图 3-1 所示。

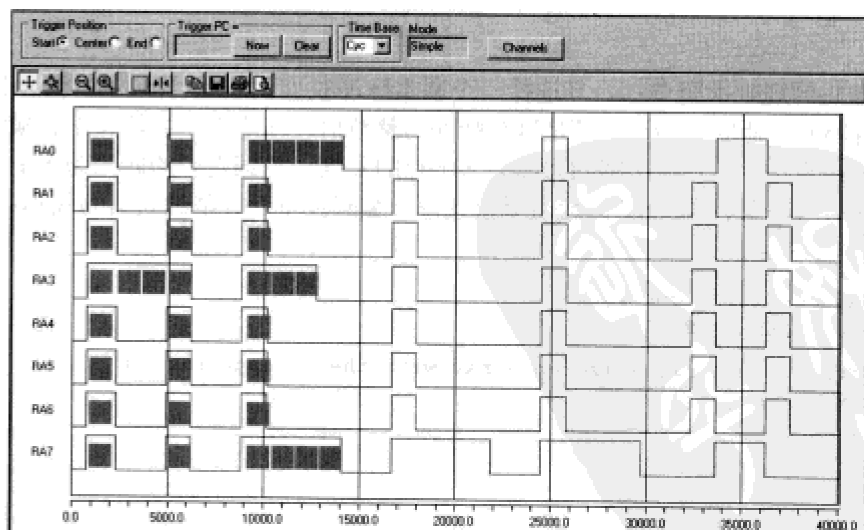


图 3-1 运行完第一次扫描后的逻辑分析仪窗口

为了帮助查看输出，图中增加了一些红点，它们代表显示序列中被点亮的 LED 灯。如果你眯着眼睛并且想象一下输出逻辑高电平的引脚那里有一个被点亮的 LED 灯，那么就能读懂该信息。

3.11 用 Explorer 16 演示板进行测试

如果你有 Explorer 16 演示板和 MPLAB REAL ICE 编程/调试器，就可以完成更有趣的实验，具体步骤如下。

（1）利用 Setup（配置）检查表配置在线调试器的选项。

（2）利用 Device Configuration（器件配置）检查表验证器件配置位是否配置成适用于 Explorer 16 演示板。

（3）利用 Programming（编程）检查表对电路内 PIC32 芯片编程。

如果将室内的照明光调弱些，当你"摇动"演示板时，就会看到该信息在闪烁，只是效果不太理想。而在仿真器和逻辑分析仪窗口中，可以选择希望看到的序列中的某一部分，并将它们"冻结"在屏幕上。而使用演示板时，你可能会发现很难将板子的移动与 LED 显示过程同步起来。

为此，可以将时间常数调整为适合你的最佳速度。经过反复实验，我发现短延时和长延时的时间常数分别取 400 和 3200 时很理想。你也可以根据自己的喜好来调整。

3.12 用 PIC32 Starter Kit 进行测试

如果你有 PIC32 Starter Kit，那么仅利用板上接在 PortD 引脚 RD0、RD1 和 RD2 上的 3 个 LED 灯验证上述程序就比较困难了，但是这并不是不可以。遗憾的是，即使你有 PID 适配板，能将 Starter Kit 与 Explorer 16 演示板连接起来，也无法圆满地看到演示，这是因为 Starter Kit 使用了 JTAG 端口，这就意味着接在 PortA 上的 8 个 LED 中，有 4 个无法使用。

这不合理。事实上，我相信能够通过改变策略找出另一种在 PIC32 Starter Kit 上向外界发送信息的方法。这个方法就是使用古老而可靠的莫尔斯电码（Morse code）！下面就是我们所需的一组闪灯序列：

```
H       E L     L     O
. . . . . . - . . . - . . - - -
```

莫尔斯电码的规则很简单：一旦选择一个基本脉冲长度代表点（大约是零点几秒），那么产生莫尔斯电码的其他间隔的长度都是这个脉冲的整数倍。横线表示 3 倍长度。横线和点之间的停顿为一个点的长度，字母之间的间隔为 3 个点的长度，单词之间的间隔为 5 个点的长度。这样，我们就能用一个包含 1、0 的数组对信息进行编码。下面就是修改后的代码：

```
#include <p32xxxx.h>

// 1. define timing constant
#define DOT_DELAY 18000

// 2. declare and initialize an array with the message bitmap
char bitmap[] = {
  // H ....
  1,0,1,0,1,0,1,0,0,0,
  // E .
  1,0,0,0,
  // L .-..
  1,0,1,1,1,0,1,0,1,0,0,0,
  // L .-..
  1,0,1,1,1,0,1,0,1,0,0,0,
  // ---
  1,1,1,0,1,1,1,0,1,1,1,
  // end of word
  0,0,0,0,0
  };

// 3. the main program
main()
```

```
{
  // 3.1 variable declarations
  int i;              // i will serve as the index

  // 3.2 initialization
  TRISD = 0;          // all PORTD as output
  T1CON = 0x8030;     // TMR1 on, prescale 1:256 PB=36MHz
  PR1 = 0xFFFF;       // max period (not used)

  // 3.3 the main loop
  while( 1)
  {
      // 3.3.1 display loop, spell a letter at a time
      for( i=0; i<sizeof(bitmap); i++)
      {
        PORTD = bitmap[i];

        // short pause
        TMR1 = 0;
        while ( TMR1 < DOT_DELAY)
        {
        }
      } // for i

  } // main loop
} // main
```

请注意，为了避免通过人工计算点和横线的个数来确定数组的长度，我在这里使用了一个技巧。只要使数组声明处的方括号（[]）内为空，就可以告诉编译器由它自己根据后面列表（花括号{}之间的部分）里的整数的个数来确定正确的数组长度。当然，如果数组的初始化列表不在数组声明处，那么这一招就没用。此外，如果我不通过其他方法获知数组内的元素个数，就会在使用 for 循环时遇到困难。幸运的是，我可以用 sizeof() 函数告诉我数组占用的字节数，由于数组的每个元素都是 char 型整数，因此我们就能准确地计算出数组的元素个数。

3.13 小结

本章回顾了一些基本变量类型（包括不同大小的整数和浮点数）的声明。在前面原创性的"摇动" LED 显示示例和后面的莫尔斯电码示例中，都用到了数组声明及其初始化，并利用 for 循环将信息发向外界。

3.14 对汇编语言行家的提示

运算符++和--实际上比你想象的要灵活得多。如果像示例中那样将它们作用于整数，那么除了能使你少敲几次键盘外没什么用处。但是，如果将它们作用于指针（这是一种存放存储器地址的变量类型），那么它将使地址改变一定数量的字节，从而指向下一个数据。例如，一个指向 16 位整数的指针会使地址增加 2，而一个指向 32 位整数的指针则会使地址增加 4，依此类推。

递增和递减运算符还能用于一般的表达式中，并在变量内容被提取之前或者之后改变变量的内容。下面是一些示例（假设初始条件为 a=0，b=1）：

```
a = b++;     // a = 1, b = 2
```

在第一个示例中，首先将 b 的值赋给 a，然后再将 b 的值递增 1。

```
a = ++b;    // a = 2, b = 2
```

在第二个示例中，首先将 b 的值递增 1，然后再将 b 的新值赋给 a。

可是，要适度地使用这些有趣的功能。要使它带来的方便性（比如减少键盘输入量）与增加代码的混乱性相平衡。

尽管看起来递增/递减运算符能潜在地增加程序的运行效率，但是这一点几乎可以忽略。事实上，无论是否使用递增/递减运算符，即使在最差的配置下，MPLAB C32 编译优化器都能够优化普通表达式中使用的 PIC32 寄存器，而无需用户考虑这些细节。

最后，还要多说几句关于循环的事情。我们可能会被众多的选择而迷惑：是该在循环的开始还是结尾处检查逻辑条件呢？是否该使用 for 循环呢？事实上，很多情况下，你要实现的算法本身就会指明该用哪个，但是很多时候程序员又有一定的自由度，因此可选的方法并不唯一。通常情况，请选用能使你的程序更具可读性的方法。如果选择哪种方法都无关紧要，比如实现主循环，那么就选择你喜欢的方法，并且保持一致。

3.15 对 PIC 单片机行家的提示

根据目标单片机的架构以及运算和逻辑单元（ALU）最终是以字节方式还是字方式进行处理，代码的紧密性和效率会有所不同。在 PIC16 和 PIC18 的 8 位架构中，要尽可能地采用字节长度的整数；而在 PIC32 架构中，处理字长度的整数与处理字节长度的整数的效率相同。而唯一限制我们在使用 MPLAB C32 编译器时不能总使用 32 位整数的因素是，单片机的片上 RAM 存储器资源相对稀缺。

3.16 对 C 语言行家的提示

尽管 PIC32 单片机的 RAM 存储器比大多数 8 位单片机的 Flash 存储器还大，但是嵌入式控制应用总是受到成本和空间的限制。如果你曾学过在 PC 或者工作站上进行 C 语言编程，那么你可能会毫不犹豫地在需要表示整数时就使用 int。但是，在嵌入式控制系统编程中就需要慎重考虑了。在程序中每次节省一字节，有时就可能使你的程序适用于空间更小的 PIC32 单片机，从而节省几十美分，再乘以成千上万个芯片（由生产速度决定），就会为公司节省大笔资金。换句话说，如果你学会使用最少的变量，那么就可能变成更好的嵌入式控制设计师。归根结底，这正是工程学的全部意义所在。

3.17 提示与技巧

第一章中我就向你介绍过神秘的启动代码（crt0），它是链接器自动在复位向量和主函数之间添加的一小段代码。而在本章中你可能还没意识到 crt0 代码又帮了我们一回。在本章开发的最后一个工程中，我们声明了一个名为 bitmap[] 的数组，并且用一组特定的数据对它进行初始化。但是，数组作为一种数据结构，它本来位于 Flash 存储器中，而在程序运行时又位于 RAM 中，这是怎么回事？事实上，正是 crt0 代码负责在主程序运行前，将数组的内容从 Flash 存储器复制到 RAM 中的。

crt0 代码完成的另一个重要功能是将程序声明的每个全局变量初始化成 0。在大多数情况下，这会使程序更加安全并且更容易预测，（难道你在使用变量前没有对它们进行初始化吗？）但是这是有代价的。如果 RAM 中要存放一个很大的数组，那么即使你没有明确地要求初始化

它们，crt0 代码也会花费一段时间将它们清零，然后才会执行主程序。然而，有些嵌入式应用是不允许出现这种延时的。在这些应用中，数毫秒都可能使一枚昂贵的功率 MOSFET 损坏(打个比方)，或者能迅速而安全地使你的应用系统从紧急复位状态恢复过来。在这些特殊的情况中，就得定义特殊函数_on_reset()，以下是一个示例：

```
void _on_reset( void)
{
  // something urgent that needs to be done immediately
  // after a reset or at power up
     your code here ...
}
```

该函数将代替 crt0 代码在开始初始化前调用的一段空白的程序空间。请注意，这段程序要尽可能短，并且不要在此做太多的假设。首先，请记住该函数会在每次 PIC32 复位后被调用。其次，不能调用除了堆栈以外的其他函数或使用全局变量，甚至不能对其进行初始化！

3.18 练习

(1) 为了提高显示和手动的同步性，可以在用手移动板之前先按下一个按钮，然后再开始显示下一个字符。

(2) 增加一个开关来检测手移动时的翻转，并且在手回扫时反向显示 LED 序列。

3.19 参考书

P. Rony、D. Larsen 和 J. Titus 所著，*The 8080A Bugbook, Microcomputer Interfacing And Programming*。这本书将我领入了单片机世界，并且永远地改变了我的生活。书中不涉及任何高级语言编程，都是讲述基本的汇编语言编程以及硬件接口方面的内容。(很遗憾，这本书现在只能在博物馆里找到了，参见下面的链接。)

3.20 链接

www.bugbookcomputermuseum.com/BugBook-Titles.html。这是 Bugbooks 博物馆的链接，Intel 8080 微处理器是 30 多年前的东西，已经成为历史。

http://en.wikipedia.org/wiki/Morse_code。这个网页介绍莫尔斯电码，包括它的历史和应用等内容。

第 4 章　算术操作与优化

4.1　计划

上一章中，我们学习了 C 语言的各种变量类型，并且特别强调了通过选取适当的变量可以节省 RAM 资源的重要性。不知各位读者现在的想法是什么，而我现在非常好奇地想知道这些变量是如何工作的，并且想看看 MPLAB C32 编译器是如何对它们进行基本算术操作的。我们知道，PIC32 单片机拥有一组 32 位“工作”寄存器以及一个 32 位 ALU，因此它的代码执行效率很高，但是我还想比较一下它对不同数据类型，特别是浮点数据类型，进行相同操作时的效率差别。希望通过本章的学习能使你更好地掌握如何在系统性能与存储资源、实时性约束及复杂度之间进行权衡，以便更好地满足嵌入式控制系统的需要。

4.2　准备

本章只使用一些软件工具，包括 MPLAB IDE、MAPLAB C32 编译器以及 MPLAB SIM 仿真器。

首先请使用 New Project Setup（创建新工程）检查表创建一个名为 NUMB3RS 的工程，并将工程的源文件命名为 NUMB3RS.c。

4.3　探索

为了了解 MPLAB C32 支持的所有数据类型，建议你看一看 MPLAB C32 User Guider（用户手册）。表 4-1 列出了编译器支持的所有整数类型。

表 4-1　MPLAB C32 的整数类型比较表

类　　型	位　　数	最　小　值	最　大　值
char, signed char	8	−128	127
unsigned char	8	0	255
Short, signed short	16	−32768	32767
unsigned short	16	0	65535
Int, signed int, long, signed long	32	-2^{31}	$2^{31}-1$
unsigned int, unsigned long	32	0	$2^{32}-1$
Long long, signed long long	64	-2^{63}	$2^{63}-1$
unsigned long long	64	0	$2^{64}-1$

ANSI C 标准共定义了 10 种整数类型，包括 char、int、short、long 以及 long long，其中每种又分别包括 signed（有符号，默认地）和 unsigned（无符号）。表 4-1 注明了 MPLAB C32 编译器为每种数据类型分配的位数，为了方便起见，还分别给出了每种数据类型所能表示的最小值和最大值。

显然，如果某个数据类型是有符号的，那么就有一位用于表示符号，从而导致所能表示的数据的绝对值减半，并且数据表示范围的中心将位于 0 附近。前面曾提到过，MPLAB C32 编

译器同等对待 int 和 long 类型，都为它们分配了 32 位（相当于 4B）。事实上，PIC32 单片机的 ALU 处理 8 位、16 位及 32 位数的效率是相同的。对这些整数类型数据进行的大部分算术和逻辑操作都可以通过一条汇编指令完成，并且该汇编指令的执行速度很快，通常只需花费 1 个时钟周期。

long long 整数类型（1999 年加入到 ANSI C 扩展集）可以表示 64 位数，共需要 8B 的存储空间。由于 PIC32 的内核是基于 MIPS 的 32 位架构的，因此编译器必须加入一段指令才能操作 long long 类型的整数。明白了这一点，我们就能预见到使用 long long 型整数时必然会损失一些性能（主要指时间开销大）；但是我们还不知道这种损失有多大。

下面看一个整型数据的示例，请输入以下代码：

```
main ()
{
int i,j,k;
  i = 1234;   // assign an initial value to i
  j = 5678;   // assign an initial value to j
  k = i * j;  // multiply and store the result in k
}
```

生成工程后（选择菜单 Project|Build All 或者按下 Ctrl+F10 组合键），就可以打开反编译窗口（选择菜单 View|Disassembly Listing），编译器产生的代码如下：

```
12:                          i = 1234;
9D00000C  240204D2  addiu        v0,zero,1234
9D000010  AFC20000  sw           v0,0(s8)
13:                          j = 5678;
9D000014  2402162E  addiu        v0,zero,5678
9D000018  AFC20004  sw           v0,4(s8)
```

即使不了解 PIC32（MIPS）的汇编语言，我们也可以很容易地看出这是两条赋值语句。它先将立即数载入寄存器 v0，接着又存入存储器中，并记作变量 i（寄存器 S8 指向它）；而后又类似地对变量 j 赋值（寄存器 S8 加上偏移量 4 后指向它）。

接下来是乘法操作部分，首先将变量 i 和 j 中的整数传回寄存器 v0 和 v1，然后执行 32 位乘法指令 mul。最后将位于 s0 中的计算结果存入变量 k(寄存器 S8 加上偏移量 8 后指向它)；多么一目了然啊！

```
14:                          k = i*j;
9D00001C  8FC30000  lw           v1,0(s8)
9D000020  8FC20004  lw           v0,4(s8)
9D000024  70621002  mul          v0,v1,v0
9D000028  AFC20008  sw           v0,8(s8)
```

注解 尽管详细分析 MIPS 汇编编程接口不在本书的讨论范围，但是我还是多说一句，相信你一定会觉得 mul 指令很有意思。和 MIPS 内核的其他算术指令一样，mul 指令也有 3 个操作数，但是在这里编译器将 v0 既作为源操作数又作为目的操作数。请注意，MIPS 内核之所以归类于 Load/Store 型体系结构，正是因为它的所有指令都是先从 RAM 提取数据并载入寄存器（load 过程），然后进行算术运算，最后又将计算结果存回 RAM（store 过程）。如果你对 MIPS 的汇编程序一点儿也不感兴趣，那么也请注意，编译器使用了 addiu 指令将立即数载入寄存器，因为这样做的效率更高。实际上它是通过将立即数加上一个名为 zero 的寄存器实现的。

4.4 关于优化（完全不优化）

你可能已经注意到，编译后的整个程序有些冗余。比如，j 的值在再次载入前（即进行乘法运算前）仍然保存在寄存器 v0 中。难道编译器没有发现这个操作是多余的吗?

事实上，编译器并不是万能的。它的任务是产生“安全的”代码，避免（至少在最开始）使用任何假设并且只使用标准指令。此后，如果使用了某些优化选项，那么第二遍（甚至是更多遍）编译将会去除这些冗余的代码。尽管如此，在工程开发和调试阶段，最好还是关闭所有的优化选项，因为它们可能修改代码的结构并可能导致无法使用单步调试或放置断点。本书其他部分也不会使用任何编译器优化选项；我们将验证，即使不使用编译器优化功能，也可以达到期望的性能。

4.5 测试

为了测试代码，我们将在反汇编列表窗口下使用软件仿真器，并单步调试每条汇编指令。或者，也可以在编辑器窗口的 C 语言源程序中，单步调试每条 C 语言语句（建议你这样做）。在这两种情况下，我们都可以执行以下步骤。

（1）打开局部变量窗口（选择菜单 View|Locals）立即查看已列出的变量的值。这个窗口虽小，但是很方便，它能列出当前函数（本例中就是 main ()）中定义的所有变量。

（2）打开观察窗口（选择菜单 View|Watch）并且利用 Add SFR 组合选择框添加（add）寄存器 v0 和 v1。

（3）单步运行（选择菜单 Debugger|Step Over 或者按下 F8 键）几行程序后，看看观察窗口内的变量的变化。正如我们在前面提到的，当观察窗口或者局部变量窗口内的变量的值发生变化时，它们会以红色高亮显示。

如果需要重复测试，可以执行复位操作（选择菜单 Debugger|Reset|Processor Reset）。然而，如果发现在第二次运行时，局部变量在被初始化前就显示出一定的数值，那么请不要惊慌。这是因为启动代码不会对局部变量（在函数内定义的变量）清零，因此，如果再次运行前未对 RAM 存储器清零，那么 RAM 中用于保存变量 i、j 和 k 的单元就会保持原先的值。

4.6 关于 long long 类型

这里，只需修改第一行的代码，就能将整个程序改为对 64 位整数变量的操作：

```
main ()
{
  long long i,j,k;

  i = 1234;    // assign an initial value to i
  j = 5678;    // assign an initial value to j
  k = i * j;   // multiply and store the result in k
}
```

重新生成该工程，并再次切换到反汇编列表窗口（如果你已经将编辑器窗口最大化，但是还没有关闭反汇编列表窗口，那么可以使用 Ctrl+Tab 组合键快速地在这两个窗口间切换），就能看到新产生的代码，它比之前的更长。尽管其初始化部分仍然一目了然，但是乘法部分则比较复杂，这里使用的指令比以前更多。

```
15:                          k = i*j;
9D00002C    8FC30000    lw         v1,0(s8)
9D000030    8FC20008    lw         v0,8(s8)
9D000034    00620019    multu      v1,v0
9D000038    00002012    mflo       a0
9D00003C    00002810    mfhi       a1
9D000040    8FC30000    lw         v1,0(s8)
9D000044    8FC2000C    lw         v0,12(s8)
9D000048    70621802    mul        v1,v1,v0
9D00004C    00A01021    addu       v0,a1,zero
9D000050    00431021    addu       v0,v0,v1
9D000054    8FC60008    lw         a2,8(s8)
9D000058    8FC30004    lw         v1,4(s8)
9D00005C    70C31802    mul        v1,a2,v1
9D000060    00431021    addu       v0,v0,v1
9D000064    00402821    addu       a1,v0,zero
9D000068    AFC40010    sw         a0,16(s8)
9D00006C    AFC50014    sw         a1,20(s8)
```

PIC32 的 ALU 每次只能处理 32 位数据，因此 64 位乘法需要执行一系列的 32 位乘法和加法。编译器在这里使用的算法与我们在小学学过的算术方法完全相同，只不过这里是一次处理 32 位数，而不是一次处理 1 位数据。实际中，为了使用 32 位指令实现 64 位乘法，需要 4 次乘法和 3 次加法，但是你可以看到，编译器实际上只使用了 3 条乘法指令。这是怎么回事呢？

事实上，两个 long long 型整数（每个都是 64 位数）相乘，会产生 128 位的结果。但是，在前面的例子中，我们已经约定要将结果保存在另一个 long long 型变量中，因此允许的结果最多只能是 64 位。我们这样做，显然会允许发生溢出（可见溢出并不是遥不可及），但同时又允许编译器安全地忽略结果的高 64 位。既然已经知道那些位将会丢失，因此编译器就省略了第四步乘法，换句话说，这已经是优化后的代码了。



注解　根据基础数学知识可知，两个 n 位宽的整数相乘，所得的结果是 $2n$ 位宽的整数。C 编译器也遵守这一点，但是如果没有准备足够的空间来存放结果，或者假如就只是没有更大的整型变量（比如两个 long long 型整数相乘就是这样），那么就只能（偷偷地）丢弃结果的高位部分。因此，我们有责任根据应用系统所需使用的变量范围选择合适的整数类型。如果有必要，可以将每个操作数的第一个非零位（即最高位）的位数之和，作为计算结果的位数。如果这个和比计算结果所用的数据类型的位数还大，那么就可以确定这里肯定会发生溢出！

4.7　整数除法

如果像前面的例子一样分析整数的除法运算，那么就能迅速发现 PIC32 处理 char、short 以及 int 型整数的除法是完全一样的。

```
main ()
{
  int i, j, k;

  i = 1234;
  j = 5678;
  k = i/j;
} // main
```

编译器产生的代码非常简单，只使用了一条汇编指令 div。

```
15:                      k = i/j;
9D00001C   8FC30000   lw         v1,0(s8)
9D000020   8FC20004   lw         v0,4(s8)
9D000024   0062001A   div        v1,v0
9D000028   004001F4   teq        v0,zero
9D00002C   00001012   mflo       v0
9D000030   AFC20008   sw         v0,8(s8)
```

但是，当分析 64 位除法时，就会发现编译器使用了完全不同的技术：

```
main ()
{
  long long i, j, k;

  i = 1234;
  j = 5678;
  k = i/j;
} // main
```

事实上，重新编译并且查看反汇编列表窗口中的新代码就会发现，这是一些容易令人误解的调用子程序的代码（jal）。

```
15:                     k = i/j;
9D000030   8FC40010   lw        a0,16(s8)
9D000034   8FC50014   lw        a1,20(s8)
9D000038   8FC60018   lw        a2,24(s8)
9D00003C   8FC7001C   lw        a3,28(s8)
9D000040   0F40001A   jal       0x9d000068
9D000044   00000000   nop
9D000048   AFC20020   sw        v0,32(s8)
9D00004C   AFC30024   sw        v1,36(s8)
```

在反汇编列表中可以查看子程序，它们位于主程序代码之后。该子程序的注释行将它清楚地与其他程序分开，并且注明该子程序来自库函数 libgcc2.c。这段程序的源代码可以在 MPLAB C32 编译器的文档中找到，它位于 MPLAB C32 编译器的安装目录中。

这里之所以要调用子程序，是因为编译器做了折中。调用子程序会增加额外的指令开销，并且增加堆栈使用量。但是另一方面，这又能使得每增加一次除法所需增加的程序空间更少，从而能够降低程序空间的总开销。

4.8　浮点数

除了整型，MPLAB C32 编译器还支持表示小数的数据类型，这就是浮点数类型。浮点数类型共有 3 种：float、double 以及 long double。根据精度的大小，可以将它们分为两组（见表 4-2）。

表 4-2　MPLAB C32 浮点数对比表

数据类型	位　数
float	32
double	64
long double	64

请注意，默认情况下，MPLAB C32 编译器为 double 和 long double 类型分配的数据位数相同，都采用了 IEEE 754 标准定义的双精度浮点型格式。

由于 PIC32 没有硬件浮点处理器（FPU），因此为实现浮点运算就必须在编译时使用浮点

运算函数库，它们比任何一个整型运算函数库都大，复杂度也最高。如果你使用了这些浮点型数据，就会发现系统性能会明显下降，但是如果调用的数量不大，那么 MPLAB C32 编译器还是能很轻松地完成任务。

下面就将前面的例子改为浮点型变量：

```
main ()
{
  float i,j,k;

  i = 12.34;        // assign an initial value to i
  j = 56.78;        // assign an initial value to j
  k = i * j;        // store the result in k
}
```

重新编译并打开反汇编列表窗口后，就会立即注意到编译器在代码中插入了子程序。

再次将程序改为使用双精度浮点数类型（long double），所得的结果与此非常类似。除了初始化赋值有些小变化，其他部分看起来一样，也是调用子程序。

由于使用 C 编译器后，使用任何数据类型都很方便，因此我们很容易习惯于使用编译器能提供的最大的整数或者浮点数类型，以确保数据运算的安全，避免出现上、下溢出。然而在嵌入式控制应用中恰恰相反，选择合适的数据类型对于平衡系统性能以及优化资源使用至关重要。为了做出一个合理的决定，我们需要了解一下选择不同数据类型对系统性能的影响情况。

4.9 评估系统的性能

下面将使用我们已经熟悉的软件仿真工具来测试 MPLAB C32 编译器使用不同算术函数库（整型的和浮点型的）的性能。我们将使用软件仿真器（MAPLAB SIM）自带的 StopWatch 工具，测试如下一段代码：

```
#include <p32xxxx.h>

main ()
{
   char          c1, c2, c3;
   short         s1, s2, s3;
   int           i1, i2, i3;
   long long     ll1, ll2, ll3;
   float         f1,f2, f3;
   long double   d1, d2, d3;

   c1 = 12;          // testing char integers (8-bit)
   c2 = 34;
   c3 = c1 * c2;

   s1 = 1234;        // testing short integers (16-bit)
   s2 = 5678;
   s3= s1 * s2;

   i1 = 1234567;     // testing (long) integers (32-bit)
   i2 = 3456789;
   i3= i1 * i2;

  ll1 = 1234;       // testing long long integers (64-bit)
  ll2 = 5678;
  ll3= ll1 * ll2;
```

```
    f1 = 12.34;         // testing single precision floating point
    f2 = 56.78;
    f3= f1 * f2;

    d1 = 12.34;         // testing double precision floating point
    d2 = 56.78;
    d3= d1 * d2;
  } // main
```

编译并链接工程后，打开 StopWatch 窗口（选择菜单 Debugger|StopWatch）并且将窗口放在自己习惯的地方（见图 4-1）。（我个人喜欢将该窗口固定在屏幕底部，这样就不会被编辑器窗口遮住，总是可见和可访问的。）

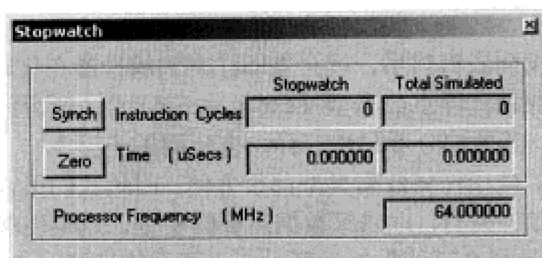


图 4-1 MPLAB SIM 的 StopWatch 窗口

单击 Zero 按钮清零 StopWatch 定时器，然后执行 Step-Over 命令（选择菜单 Debug|StepOver 或者按 F8 键）。仿真器更新 StopWatch 窗口后，就可以手动记录整数运算花费的时间。该时间是由仿真器根据时钟周期计算得到的，它等于时钟数除以仿真所用的时钟频率（该参数可在调试器设置对话框中设置，选择 Debugger|Settings|Osc/Trace 选择卡即可）。

接下来将光标移到下一条乘法，执行 Run to Cursor 命令；或者反复执行 StepOver 命令直到执行完乘法操作。然后，再次清零 StopWatch，并执行一条 StepOver 命令。如此反复。直到 5 种数据类型的乘法都测试完毕。测试结果见表 4-3。

表 4-3 基于 MPLAB C32 rev. 0.20 的性能对比测试结果（关闭了所有的优化功能）

乘法测试	宽度（位数）	时钟周期数	性能对比	
			整型	浮点型
字符型整数（char）	8	6	1	—
短整型（short）	16	6	1	—
整型（int，long）	32	6	1	—
长整型（long long）	64	21	3.5	—
单精度浮点数（float）	32	71	11.8	1
双精度浮点数（long double）	64	159	26.5	2.23

表 4-3 的第一列记录了测试结果（时钟数），后面两列显示了相对性能比（每行的时钟数分别除以两个参考类型花费的时钟数）。如果测得的结果与表中的不同，也不必担心，因为影响测试结果的因素很多。高版本的编译器可能使用效率更高的函数库，或者在测试时打开了优化功能。

请注意，这种测试不像真正的性能测试平台那么严格。我们在这里所做的只是想说明使用

不同的数据类型进行计算会改变系统的性能。由此看来，表 4-2 足以为我们证明这一点。

正如所料想的那样，32 位运算速度很快，然而 long long 型整数（64 位）的乘法的运算速度仅有 32 位的四分之一。单精度浮点数乘法比整数乘法的时间开销更大。32 位浮点数相乘需要的时间开销差不多是 32 位整数乘法的 12 倍。而双精度浮点数乘法（64 位）的时间开销则是单精度浮点数乘法的 2 倍。

那么，我们何时该用浮点数，何时又该用整数呢？

从已掌握的有限知识来看，目前可以总结出以下规则。

（1）尽可能使用整数，即当不需要小数或者算法可由整数运算实现时，就用整数。

（2）如果希望节省 RAM 存储器空间，那么请使用不会溢出的位数最少的整数。但是，如果不使用 64 位整数，那么不论使用哪种小于 32 位的整数，都看不出任何性能的提升。

（3）如果必须使用浮点数（比如需要使用小数），那么需要考虑在性能上可能会下降 10 倍。

（4）双精度浮点数（long double 型）看起来只会进一步降低性能，它比单精度浮点数的性能还低一倍。

请记住，尽管浮点数具有最大的数据表示范围，但是它总存在一定的误差。因此，在商业计算中不推荐使用浮点数。如果有必要，请使用 long long 型整数，并且将所有的运算都基于分（而不是元和由此产生的小数）。

4.10 小结

在本章中我们不仅学习了 MPLAB C32 支持的数据类型，以及各种数据类型分别需要的存储空间，而且还了解了它们对编译后的程序代码体积及运行速度的影响。我们利用 MPLAB SIM 仿真器的 StopWatch 工具测试了执行一系列基本算术运算所需的指令周期数。所总结的部分知识对于将来在嵌入式应用系统的精度和性能之间进行折中具有重要意义。

4.11 对汇编语言行家的提示

那些极少数敢于处理浮点数的汇编语言行家肯定非常高兴并且永远感谢 C 编译器带来的巨大便利。在 C 语言编程中，单精度和双精度运算变得和整数运算一样简单。

尽管如此，使用整数有时仍然可能会失控，这是因为编译器隐藏了运算实现的细节，并且有些运算看起来难以理解或者不太直观（可读性差）。下面是一些数据类型变换和字节操作运算可能带来的问题：

- ❑ 将整型数转换为更小或者更大的整型数；
- ❑ 提取或者设置 16 位或者 32 位数据的最高位/最低位；
- ❑ 提取或者设置整数变量的某一位。

C 语言可以方便地通过隐式转换完成上述转换，比如：

```
short  s;        // 16-bit
int    i;        // 32-bit
i = s;
```

变量 s 的值会被送入变量 i 的低 2 位，而 i 的高 2 位会被清零。

隐式转换（也称为强制类型转换）用于直接赋值可能引起编译器报错的场合，比如：

```
short  s;        // 16-bit
int    i;        // 32-bit
s = (short) i;
```

(short) 是一个强制类型转换，会丢弃变量 i 的高 2 位，从而使变量 i 成为 16 位整数。

当待变换的整数的位宽小于 1B 时，就由位场添补。PIC32 的函数库文件包括大量用于操作外围设备的特殊功能寄存器的控制位的定义。

下面是一些从本工程所使用的头文件中提取出来的位场定义的例子，其中定义了 Timer1 的控制寄存器 T1CON，它的每个控制位都可通过结构体 T1CONbits 进行访问。

```
extern unsigned int T1CON;
extern union {
  struct {
    unsigned :1;
    unsigned TCS:1;
    unsigned TSYNC:1;
    unsigned :1;
    unsigned TCKPS0:1;
    unsigned TCKPS1:1;
    unsigned TGATE:1;
    unsigned :6;
    unsigned TSIDL:1;
    unsigned :1;
    unsigned TON:1;
  };
  struct {
    unsigned :4;
    unsigned TCKPS:2;
  };
} T1CONbits;
```

利用“点”符号就可以访问每个位场，比如：

```
T1CONbits.ON = 1;
```

4.12 对 8 位 PIC 单片机行家的提示

熟悉 8 位 PIC 单片机及其编译器的用户将会发现，PIC32 的整数和浮点数运算性能都有显著提高。PIC32 配备的 32 位 ALU 在每个周期可以对数据位操作 4 次，此外，多达 32 个的工作寄存器还能进一步提高其性能，它们使关键运算程序和大规模运算的代码效率显著提高。

4.13 对 16 位 PIC 和 dsPIC 单片机行家的提示

MPLAB C30 编译器的用户可能已经注意到，到目前为止，新的 MPLAB C32 编译器将普通的整型定义成多种位宽。比如，int 和 short 型在 MPLAB C30 中都被同样定义为 16 位。然而在 MPLAB C32 中，尽管 short 仍然是 16 位宽，但是 int 现在则拓宽成长整型。换句话说，int 的位宽增加了一倍。你可能想知道这会对代码的移植带来怎样的影响。

答案取决于如何看待这个问题。如果要将代码向“上”移植，换句话说，要将原先面向 16 位单片机编写的代码移植到 32 位 PIC 单片机时，那么很可能会顺利完成。移植后，只是全局变量使用的存储器空间会增大，堆栈的使用量也会增加，但是所使用的 PIC32 单片机也会提供更大的 RAM 存储器空间。由于新型的整型比原来代码使用的空间大，因此只要编写合理，那么就不必担心发生上溢出和下溢出。

相反，如果要将代码向“下”移植，那么即使这只是一个打算，也要慎重。如果是面向 PIC32 编写代码，并且使用 32 位的 int 型，那么当使用 MPLAB C30 对该代码重新编译时，你可能会大吃一惊。为了避免对所使用的整数的宽度含混不清，最好还是使用对应宽度的整型。

函数库 Inttypes.h 中包括了一段特殊的子集，用于定义精确宽度的整型，具体如下：

`int8_t`	总代表 8 位有符号整数
`uint8_t`	总代表 8 位无符号整数
`int16_t`	总代表 16 位有符号整数
`uint16_t`	总代表 16 位无符号整数
`int32_t`	总代表 32 位有符号整数
`uint32_t`	总代表 32 位无符号整数
`int64_t`	总代表 64 位有符号整数
`uint64_t`	总代表 64 位无符号整数

如果在需要的时候使用它们，由于它们突出显示了你所编写的代码中与整数大小有关的部分，因此这将提升代码的可移植性和可读性。



注解 另一个重要但又常被误解的整数类型是 size_t，它定义在 stddef.h 库函数中。当需要保存存储器中多个字节长度的对象时，就可以使用它。ANSI 编译器将确保它的容量能保存运算中可能出现的最大值。正如你所期望的那样，string.h 库中的 sizeof() 及其他函数都使该类型得以广泛使用。

4.14 提示与技巧

4.14.1 数学函数库

MPLAB C32 编译器支持以下的标准 ANSI C 函数库。

- limit.h 包含很多有用的宏，定义了一些与具体实现相关的约束。比如组成 char 类型的位数（CHAR_BIT）或者最大的整数（INT_MAX）。
- float.h 与 limit.h 类似，包含了适用于浮点数的与具体实现相关的约束的宏定义。比如，单精度浮点数的最大指数（FLT_MAX_EXP）。
- math.h 包含了三角函数、随机函数、对数函数以及指数函数，此外还有很多重要的常数定义，比如π（M_PI）。

4.14.2 复数数据类型

MPLAB C32 编译器支持复数数据类型，以作为整数和浮点数类的扩展。下面是一个单精度浮点型复数的声明：

```
__complex__ float z;
```



提示 关键字 complex 前后分别有两个下划线。

变量 z 现在就有了实部和虚部，它们可以单独使用，对应的语法是：

```
__real__ z
```

和

```
__imag__ z
```

类似地，下面的声明产生了一个 32 位整型复数：

```
__complex__ int x;
```

只要使用前缀 i 或者 j，就可以定义复常数，比如：

```
x = 2 + 3j;
z = 2.0f + 3.0fj;
```

所有的标准运算（+、-、*、/）对复数依然有效。此外，利用~运算符可以求得复数的共轭。

复数类型在有些应用中非常方便，它能提高代码的可读性，并有利于防止普通错误。遗憾的是，目前，MPLAB IDE 在调试时还无法全面支持复数变量，只能通过观察窗口或者鼠标悬停查看复数的实部。

4.15 练习

（1）编写一段代码，使用 Timer2 作为 stopwatch，进行实时的性能测试。

（2）如果 Timer2 的宽度不够，就让 Timer2 和 Timer3 联合工作在 32 位定时器方式，实现上述功能。

（3）测试不同数据类型的除法运算的相对性能。

（4）测试三角函数与普通运算的性能对比。

（5）测试复数乘法的相对性能。

4.16 参考书

Robert Britton 所著的 *MIPS Assembly Language Programming*。我在这里推荐一本汇编语言编程方面的书，好像有点儿奇怪。当然，本书主要研究 C 语言编程，但是如果你和我一样，那么就忍不住好奇，也想学习 PIC32 MIPS 内核的汇编语言。

4.17 链接

http://en.wikipedia.org/wiki/Taylor_series。如果你对泰勒级数感兴趣，那么访问这个网站就可以了解 C 编译器的数学函数库是如何近似一些函数的。

第 5 章　中　　断

5.1　计划

受效率、尺寸以及最终成本等因素的影响，嵌入式控制领域中最简单的应用系统，其产量往往也最大，但它们大都无法实现“奢华”的多任务操作，因此转而采用中断，以便在多任务间实现“分身术”。中断是实时控制中的重要技术，它使得应用系统能够处理异步的外部事件。但是，C 语言模型中并没有中断的概念，因此嵌入式控制系统的程序员只能将中断定义成一种特殊的函数。

在本章中，我们将学习如何使用 MPLAB C32 编译器轻松地管理 PIC32 单片机提供的各种中断机制。

5.2　准备

本章仅使用软件工具，包括 MPLAB IDE、MPLAB C32 编译器以及 MPLAB SIM 仿真器。

首先请使用 New Project Setup（创建新工程）检查表创建一个名为 Interrupts 的工程以及一个名为 interrupts.c 的源文件。

5.3　探索

中断是指一种要求迅速引起 CPU 注意的内部或外部事件。PIC32 单片机具有资源丰富的中断系统，它能管理多达 64 种不同的中断源。如果有必要，每个中断源都直接对应了唯一的一段代码，即所谓的 ISR（Interrupt Service Routine，中断服务程序）或者*中断处理程序*（interrupt handler），它能完成期望的响应动作。中断可以和主程序的执行流程完全异步。它们能够在任何时刻以任意顺序被触发。

快速响应中断对于迅速回应触发事件并返回主程序执行流程极为重要。因此，从触发事件至执行中断服务程序第一行代码之间的时间[即所谓的*中断潜伏期*（interrupt latency）]要尽可能短。PIC32 单片机的中断潜伏期极短。尽管每个中断源的潜伏期都是固定的，只有三四个指令周期，但是 32 位架构中的其他模块，比如后面将详细介绍的 cache（高速缓存）以及总线仲裁模块，都会影响总响应时间，从而使中断潜伏期出现一些不确定性。深入理解中断原理有助于最大限度地减小或者消除上述因素对应用系统的影响。

MPLAB C32 编译器提供的一些语言扩展以及函数库 plib.h 中的大量函数有助于我们管理 PIC32 单片机复杂的中断系统。

5.4　中断和异常

对于 PIC32 单片机内运行的 MIPS 内核来说，所有的中断都可以看作*异常*（exception）。异常是指所有能够打断程序正常执行流程的事件，因此涵盖的范围很宽。比如，复位命令就能产生异常，除法错误也会产生异常，访问无效（或者受限）的存储器地址也会产生异常，等等。而中断则是一种最无危害的异常。MIPS 内核依靠位于 RAM、程序存储器或者同时位于二者中的*向量*（即函数指针）控制几乎所有的异常（参见表 5-1）。启动代码负责配置中断向量，并为嵌入式控制应用系统可能需要使用的所有重要异常提供默认的中断处理程序。

如果你看不懂表 5-1 中的中断向量，那么也不必担心。表中包含的高级功能将在后面的章节中进行讨论和学习。虽然有些功能属于 MIPS 架构，但是在 PIC32MX 中并未实现。

表 5-1　PIC32 架构的异常向量表

异　常　源	存　储　器	描　　述
Reset 和 NMI	程序存储器	普通复位和非屏蔽中断入口
片上调试	程序存储器	ICD 和 EJTAG 结构使用它启动在线调试功能
Cache（高速缓存）错误	RAM 或程序存储器	Cache 的错误条件
TLB（旁路转换缓冲）重装	RAM 或程序存储器	由于 PIC32 采用固定地址变换机制（FMT）来替代完整的 MMU 模块，因此不会出现该异常
普通异常	RAM 或程序存储器	所有其他类型的异常
中断	RAM 或程序存储器	合适的中断向量

由于基本的 MIPS 中断在异常表中仅对应唯一的向量，因此所有的中断事件都对应同一个中断复位程序。一旦发生中断（异常），特殊寄存器（就是所谓的起因）就会为服务程序提供能识别触发事件并做出最合适响应所需的所有信息。为了能够在处理完中断后继续运行主程序，中断服务程序必须先保存处理器的上下文（序言），然后才能执行其他操作，当中断服务程序结束时，又要恢复上下文（结尾）。精确的序言和结尾执行过程有时是缠绕在一起的，对它们的分析已经超出了本书的研究范围。而现在，只需要知道 MPLAB C32 编译器能够自动并且安全地完成上述工作，并且允许用户定义“特别的”C 函数作为中断服务程序即可。此外，还需要注意以下一些限制。

- 中断服务程序不能返回任何数据（函数类型为 void 型）。
- 中断服务程序不能传递参数（参数类型为 void 型）。
- 其他函数无法直接调用中断服务程序。
- 理想情况下，中断服务程序最好不要调用其他函数。

前 3 个限制充分体现了中断机制的本质：中断是由异步事件触发的。由于最开始并没有任何函数调用中断服务例程，因此中断服务程序不可能传递参数，也没有返回值。最后一条限制主要是从确保执行效率的角度提出的建议。

5.5　中断源

下列事件可以触发中断。PIC32FJ512MX360L 的外部中断源包括：

- 5 个具有电平触发检测功能的外部引脚；
- 22 个与变更通知（Change Notification）模块相连的外部引脚；
- 5 个输入捕获模块；
- 5 个输出比较模块；
- 2 个串行通信接口（UART）；
- 4 个同步串行通信接口（SPI 和 I^2C）；
- 1 个并行主控端口。

其内部中断源包括：

- 1 个 32 位内部（核心）定时器；
- 5 个 16 位定时器；

- ❑ 1个模数转换器；
- ❑ 1个模拟比较模块；
- ❑ 1个实时时钟和日历模块；
- ❑ 1个Flash控制器；
- ❑ 1个故障导向安全的时钟监视器；
- ❑ 2个软中断；
- ❑ 4个DMA通道。

其他型号的PIC32单片机具有不同的内部和外部中断源组合。很多中断源又能产生不同的中断。比如，串行通信接口（UART）就能产生3种中断：

- ❑ 当接收到新数据，并将数据放入接收缓冲区等待处理时；
- ❑ 当发送缓冲区中的数据被发送后，缓冲区变空并准备好下一次发送时；
- ❑ 当产生错误并且为了确保稳定通信需要执行必要的操作时。

通过设计，PIC32单片机的中断控制模块可以管理多达96种独立的事件。需要管理的中断可真多啊！

当然，当应用系统需要使用多个中断源时，中断服务程序就需要识别出正确的中断源，并且跳转到对应的代码段进行处理。我们很快就将看到，为了完成这项任务，程序员还需要使用一些标志位和控制功能。

5.6 中断优先级

每个中断源都有一些相关的控制位，它们按照逻辑成组地分布于各特殊功能寄存器中。

- ❑ 中断使能位（器件的数据手册中通常在中断源外围设备名称后面增加IE后缀），只包含1位数据：
  - ■ 该位被清零时，触发事件无法产生中断；
  - ■ 该位被置位时，允许处理中断。

  上电时，默认情况下所有的中断源都是关闭的。
- ❑ 中断标志位（通常带有后缀IF），只包含1位数据。每次触发事件被激活时，该位被置位，而与中断使能位的状态无关。请注意，中断标志位一旦被置位，就必须由用户（手动）清除。换句话说，必须在退出中断服务程序前对该位清零，否则该中断服务程序会再次被立即调用。
- ❑ 组优先级（通常带有后缀IP）。中断共有7个优先级（从ipl1至ipl7）。当同一时刻发送两个中断时，优先级高者首先被响应。中断源的优先级由3个数据位编码决定。无论何时，PIC32的运行优先级都保存在MIPS内核状态寄存器中。优先级比当前PIC32的运行优先级低者都无法被响应。上电时，所有中断源的优先级都被默认地设定为ipl0，并屏蔽所有的中断。
- ❑ 子优先级。在相同的优先级组内，还有两个数据位用于定义4个子优先级。如果有2个优先级相同的事件同时发生，那么子优先级高者将先被响应。一旦选择了某个优先级组里的一个中断，那么同组内的其他中断（即使是子优先级最高者）都必须等到当前中断（标志位）清零后才能被响应。

每款PIC32单片机都定义了各种中断源的（默认的）相对优先级。当其他条件都无效时（比如组优先级和子优先级都相同时），将根据自然顺序决定响应同时发生的多个事件中的哪一个（参见表5-2）。

表 5-2 PIC32FJ512MX360L 的中断源

自然顺序	宏缩写	IRQ 标志	描 述
0（最高）	CT	_CORE_TIMER_IRQ	内核定时器中断
1	CS0	_CORE_SOFTWARE_0_IRQ	内核软中断 0
2	CS1	_CORE_SOFTWARE_1_IRQ	内核软中断 1
3	INT0	_EXTERNAL_0_IRQ	外部中断 0
4	T1	_TIMER_1_IRQ	定时器 1 中断
5	IC1	_INPUT_CAPTURE_1_IRQ	输入捕获器 1 中断
6	OC1	_OUTPUT_COMPARE_1_IRQ	输出比较器 1 中断
7	INT1	_EXTERNAL_1_IRQ	外部中断 1
8	T2	_TIMER_2_IRQ	定时器 2 中断
9	IC2	_INPUT_CAPTURE_2_IRQ	输入捕获器 2 中断
10	OC2	_OUTPUT_COMPARE_2_IRQ	输出比较器 2 中断
11	INT2	_EXTERNAL_2_IRQ	外部中断 2
12	T3	_TIMER_3_IRQ	定时器 3 中断
13	IC3	_INPUT_CAPTURE_3_IRQ	输入捕获器 3 中断
14	OC3	_OUTPUT_COMPARE_3_IRQ	输出比较器 3 中断
15	INT3	_EXTERNAL_3_IRQ	外部中断 3
16	T4	_TIMER_4_IRQ	定时器 4 中断
17	IC4	_INPUT_CAPTURE_4_IRQ	输入捕获器 4 中断
18	OC4	_OUTPUT_COMPARE_4_IRQ	输出比较器 4 中断
19	INT4	_EXTERNAL_4_IRQ	外部中断 4
20	T5	_TIMER_5_IRQ	定时器 5 中断
21	IC5	_INPUT_CAPTURE_5_IRQ	输入捕获器 5 中断
22	OC5	_OUTPUT_COMPARE_5_IRQ	输出比较器 5 中断
23	SPI1E	_SPI1_ERR_IRQ	SPI 1 故障
24	SPI1TX	_SPI1_TX_IRQ	SPI 1 发送成功
25	SPI1RX	_SPI1_RX_IRQ	SPI 1 接收成功
26	U1E	_UART1_ERR_IRQ	UART 1 错误
27	U1RX	_UART1_RX_IRQ	UART 1 接收中断
28	U1TX	_UART1_TX_IRQ	UART 1 发送中断
29	I2C1B	_I2C1_BUS_IRQ	I2C 1 总线冲突事件
30	I2C1S	_I2C1_SLAVE_IRQ	I2C 1 从机事件
31	I2C1M	_I2C1_MASTER_IRQ	I2C 1 主机事件
32	CN	_CHANGE_NOTICE_IRQ	输入电平变化中断
33	AD1	_ADC_IRQ	ADC 转换结束中断
34	PMP	_PMP_IRQ	并行主模式接口中断
35	CMP1	_COMPARATOR_1_IRQ	比较器 1 中断

（续）

自然顺序	宏缩写	IRQ 标志	描　述
36	CMP2	_COMPARATOR_2_IRQ	比较器 2 中断
37	SPI2E	_SPI2_ERR_IRQ	SPI 2 故障
38	SPI2TX	_SPI2_TX_IRQ	SPI 2 发送成功
39	SPI2RX	_SPI2_RX_IRQ	SPI 2 接收成功
40	U2E	_UART2_ERR_IRQ	UART 2 错误
41	U2RX	_UART2_RX_IRQ	UART 2 接收中断
42	U2TX	_UART2_TX_IRQ	UART 2 发送中断
43	I2C2B	_I2C2_BUS_IRQ	I2C 2 总线冲突事件
44	I2C2S	_I2C2_SLAVE_IRQ	I2C 2 从机事件
45	I2C2M	_I2C2_MASTER_IRQ	I2C 2 主机事件
46	FSCM	_FAIL_SAFE_MONITOR_IRQ	故障导向安全的时钟监视器中断
47	RTCC	_RTCC_IRQ	实时时钟中断
48	DMA0	_DMA0_IRQ	DMA 通道 0 中断
49	DMA1	_DMA1_IRQ	DMA 通道 1 中断
50	DMA2	_DMA2_IRQ	DMA 通道 2 中断
51	DMA3	_DMA3_IRQ	DMA 通道 3 中断
……			
56（最低）	FCE	_FLASH_CONTROL_IRQ	Flash 控制事件

5.7　中断服务程序的声明

MPLAB C32 编译器支持两种方式将函数声明为具有一定优先级（比如 ipl1）的中断服务程序（比如 vecter 0），第一种方法是使用 *attribute* 语句，如：

```
void __attribute__ (( interrupt(ipl1),vector(0)))
InterruptHandler( void)
{
  // your interrupt service routine code here. . .
} // interrupt handler
```

第二种方法是使用 pragma 语句，如：

```
#pragma interrupt InterruptHandler ipl1 vector 0
void InterruptHandler( void)
{
  // interrupt service routine code here. . .
} // interrupt handler
```

这两种方法的结果都是使编译器将函数 InterruptHandler() 看作中断服务程序，并且自动地完成保护现场和恢复现场。

MPLAB C32 编译器在这里以及很多其他情况下都使用_attribute_(()) 语句定义特殊属性，在改变编译器行为的同时又不违反 C 语言的语法。在我看来，该语句过于隐蔽；attribute

之前和之后的两个下划线以及两组圆括号令我眼花缭乱。我喜欢使用宏(通常定义在 sys/attribs.h 文件中)来声明中断，它们看起来很像以前的 PIC24 和 dsPIC 的函数库里的声明，比如：

```
_ISR( v, ipl)
```

在下面的示例中，采用宏__ISR 声明中断，其功能与前面两段代码相同：

```
void __ISR( 0, ipl1) InterruptHandler (void)
{
  // interrupt service routine code here. . .
} // interrupt handler
```

你可以根据自己的喜好和习惯来选择这两种语句。此外，一旦需要将代码移植到不同的编译器时，就会发现其中某一种声明方法与原先的代码更接近。因此这两种方法都要学习，只有实际使用时才知道哪种方法更有用。

5.8 管理中断的函数库

PIC32 的中断源多达 96 种，为了管理 PIC32 中断控制器模块提供的复杂优先级机制，我们当然需要一些帮助，比如使用 PIC32 标准工具包里的函数库 int.h。

我们可以用以下方式直接调用它：

```
#include <int.h>
```

或者间接地将它作为整个外围设备支持函数库的一部分，如：

```
#include <plib.h>
```

经过上述两种方式的定义后，就能调用很多前面提到的函数和宏定义（宏定义以小写 m 作为前缀），具体包括如下几次。

- INTEnableSystemSingleVectoredInt();。该函数按照严格顺序（根据器件数据手册的要求）对中断控制模块进行初始化，以启动 PIC32 的基本中断管理模式。尽管该函数的名字很长，但是使用它很值得，因为它能使代码更简单、更可靠，从而显著缓解我们的负担。
- mXXSetIntPriority(x);。该函数实际上是一组类似的宏定义的占位符（将 XX 换为表 5-2 中的每个宏名称缩写即可）。它能指定所选中断源的优先级（从 0 至 7）。尽管看起来该函数的工作量不是很大，但是它带来很多便利，使我们不必痛苦地在器件数据手册中查找与所选中断源对应的 IPCxx 寄存器，选择其中的-IP 位选择的对应的中断源即可。
- mXXClearIntFlag();。这是一个宏，它也代表了一组宏定义，可以清除所选中断源的中断标志（-IF 位）。

5.9 单向量中断的管理

下面将介绍定时器中断服务程序的示例。我们将启动定时器 2 模块，将其计数周期设为 15 并且要求它产生中断。在每个定时周期内，中断服务程序会使全局变量 count 增加 1。

```
/*
** Single Interrupt Vector test
*/
#include <p32xxxx.h>
#include <plib.h>
```

```
int count;

#pragma interrupt InterruptHandler ipl1 vector 0
void InterruptHandler( void)
{
  count++;
  mT2ClearIntFlag();
} // Interrupt Handler

main()
{
  // 1. init timers
  PR2 = 15;
  T2CON = 0x8030;

  // 2. init interrupts
  mT2SetIntPriority( 1);
  INTEnableSystemSingleVectoredInt();
  mT2IntEnable( 1);

  // 3. main loop
  while( 1);
} // main
```

每个中断服务程序（无论它有多么简单）都必须完成一个基本操作，那就是在返回前清除中断标志位。这也是我们设计的中断服务程序在为 count 加 1 之外必须完成的工作。

此外，请注意 main() 函数，在初始化定时器的控制寄存器和周期寄存器（//1.）之后，必须先完成中断配置（//2.），然后才能打开中断源。此外，定时器 2 的中断优先级（1）必须与#pragma 语句（ipl1）声明的优先级相同。



注解 编译器必须了解中断服务程序的优先级，以便使用正确的序言和结尾。事实上，我们很快就会了解到，中断 ipl7 的序言和结尾更短，这是因为它使用了别的寄存器以便实现快速切换。

如果不使用 int.h 函数库，而是直接访问负责配置中断控制器的特殊功能寄存器，那么编写代码的困难就更大。

```
/*
** Single Interrupt vector test
*/
#include <p32xxxx.h>

#define _T2IE IEC0bits.T2IE
#define _T2IF IFS0bits.T2IF
#define _T2IP IPC2bits.T2IP

int count;

void __ISR( 0, ipl1) InterruptHandler( void)
{
  count++;
  _T2IF = 0;
} // interrupt handler

main()
{
  // 1. init timers
```

```
    PR2 = 15;
    T2CON = 0x8030;

    // 2. init interrupts
    _T2IP = 1;
    INTEnableSystemSingleVectoredInt();
    _T2IE = 1;

    // 3. main loop
    while( 1);
} // main
```



对 PIC24 和 dsPIC 行家的提示 MPLAB C32 编译器的标准头文件中并不包含 PIC24 和 dsPIC 单片机标准头文件中定义的缩写_T2IF、_T2IE 以及_T2IP 等。如果需要移植 16 位代码并要求保持兼容，那么就请依照我的示例，手动逐个重新定义需要使用的缩写。

这也是个人喜好的问题。请随意选择你喜欢的方式或者是你觉得更直观、更易读的方法。

下面就请准备好新工程，进行中断测试。

(1) 将源文件保存为 single.c，然后用 New Project Setup（创建新工程）检查表建立名为 single.mcp 的工程，再将源文件加入新工程。

(2) 使用 MPLAB SIM Setup（MPLAB SIM 配置）检查表准备好 MPLAB SIM 仿真器，作为调试工具。

(3) 使用 Project|Build 命令（或者按 Ctrl+F10 组合键）生成工程。

(4) 打开 Watch 窗口（选择菜单 View|Watch）并且添加全局变量 count，在组合框中选择它，并单击 Add Symbol 按钮。

(5) 在 SFR 组合框中选择 TMR2 寄存器，然后单击 Add SFR 按钮将它添加至 Watch 窗口。

(6) 在中断服务程序中 count 变量递增的那一行放置断点，然后选择 Animate（或者 Run）开始执行代码。

如果一切正常，那么过一会儿程序就会暂停在中断服务程序中的断点处。尽管程序会陷入主循环中，但是一旦导致定时周期（PR2 寄存器的设定值）溢出，那么定时器 2 就会发出中断请求，然后将控制权移交给中断服务程序。

继续动态运行（或者再次运行）一会儿，就会发现当主循环每次被“打断”后，count 都会递增 1。

请注意，每次到达断点时，窗口内的 count 值会立即更新并且以红色显示（因为它一直在变化），但是 TMR2 的值则保持不变，你可能会惊讶地发现它并不是零。事实上，当定时器 2 计数值到达周期寄存器（PR2）的设定值时，定时器会被复位，并产生一个新的中断，但是当 PIC32 开始执行中断服务程序时，定时器又将开始计数。当中断服务程序完成现场保护并运行到断点时，定时器 2 已经显示为 2。不知不觉中，我们已经粗略地测试了中断服务程序保护现场的时间开销。由于我们为定时器 2 输入时钟选择的预分频系数为 1:8，因此计数值 2 表明中断服务程序保护现场的时间为 16 个时钟周期，相当于执行了 16 条指令。如果你感兴趣，可以在反汇编窗口中查看编译器生成的代码。Single.c 工程的屏幕截图见图 5-1。

然而，如果选择的预分频系数没有这么大（1:8）或者中断周期更短，那又会发生什么情况呢？你可以对上述代码稍做修改后自己测试一下。你将看到中断服务程序将被反复调用，从而

无法在主循环中继续运行。尽管这对我们的简单示例影响不大，但是在实际的应用系统中就会带来灾难。当中断太多、太频繁或者无法控制时，主程序就会完全停滞。因此必须确保中断服务程序，包括它的序言和结尾，都不会用光可用的处理器周期。

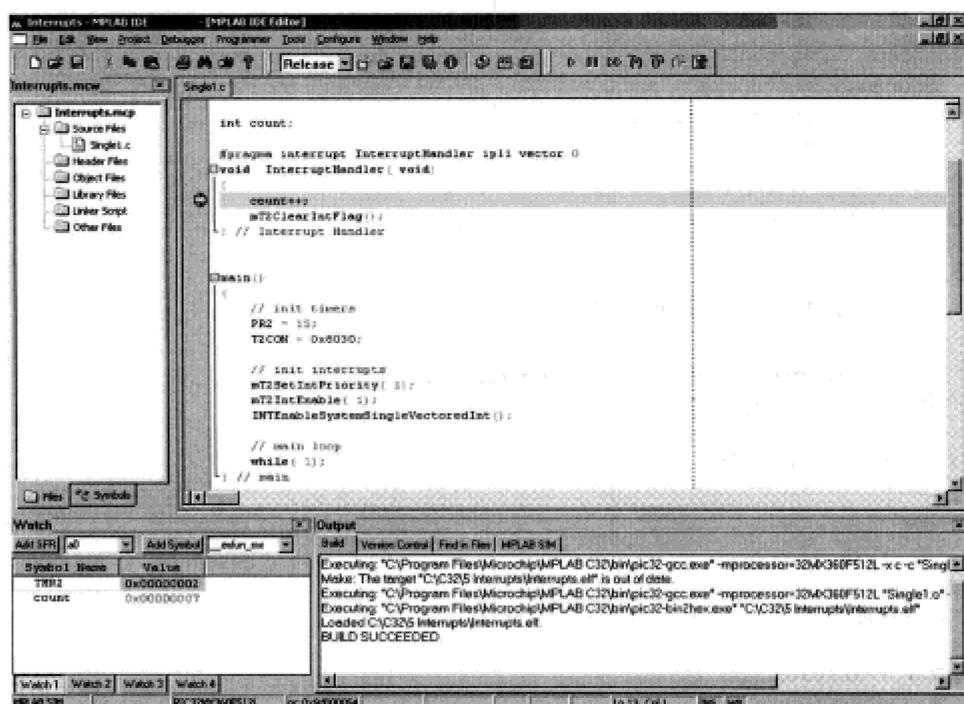


图 5-1 single.c 工程的屏幕截图

5.10 管理多个中断

如果应用程序使用多个中断，那么为中断指定不同的优先级只能解决部分问题。中断优先级将决定当两个或多个中断同时发生时，哪个最先被响应。但是，当正在处理某个中断时，其他待响应的中断就必须等待。然而，应用系统有时不仅要求多中断，而且还要求中断嵌套。当中断服务程序正在处理某个低优先级的中断时，还能立即响应高优先级的中断，也就是说允许打断中断服务程序。

为了能够嵌套中断，必须在进入中断服务程序后重新“手动”使能中断（可使用 MIPS 汇编指令），而不像以往那样在中断服务程序的末尾处才使能中断。

下面是将我们的第一个示例扩展后的虚构的应用，其中定时器 3 用于产生第二个周期中断，其优先级更高，为 3 级。

```
/*
** Single Vector Interrupt Nesting
*/
#include <p32xxxx.h>
#include <plib.h>

int count;

void __ISR( 0, ipl1) InterruptHandler( void)
```

```
{
  // 1. re-enable interrupts immediately (nesting)
  asm("ei");

  // 2. check and serve the highest priority first
  if ( mT3GetIntFlag())
   {
     count++;
     // clear the flag and exit
     mT3ClearIntFlag();
   } // _T3

   // 3. check and serve the lower priority
   else if ( mT2GetIntFlag())
   {
     // spend a LOT of time here!
     while( 1);

     // before clearing the flag and exiting
     mT2ClearIntFlag();
   } // _T2
 } // Interrupt Handler

 main()
 {
   // 4. init timers
   PR3 = 20;
   PR2 = 15;
   T3CON = 0x8030;
   T2CON = 0x8030;

   // 5. init interrupts
   mT2SetIntPriority( 1);
   mT3SetIntPriority( 3);
   INTEnableSystemSingleVectoredInt();
   mT2IntEnable( 1);
   mT3IntEnable( 1);

   // main loop
   while( 1);
 } // main
```

请注意，在//1.部分中，MIPS 汇编指令 ei 允许立即响应中断并跳转到中断服务程序。省略这一行代码将使中断自动排队并且等待后续响应。

接下来，在//2.部分中，我们首次使用了 int.h 库中的宏 mT3GetIntFlag()。顾名思义，它是用来测试定时器 3 的中断标志的。由于允许多个中断，因此我们需要通过测试来确定到底是谁引起的中断。首先将测试优先级最高的中断，然后再处理//3.部分中的优先级较低的中断，直到所有已打开的中断源都被测试完毕。

为了生成并测试新代码，需要完成以下步骤。

(1) 将新代码保存为 nesting.c，然后根据相关的检查表将该文件添加至工程中。

(2) 将 single.c 从工程中删除。

(3) Build（生成）工程。

(4) 在 count 递增那一行放置 breakpoint（断点）。

(5) 在 Watch 窗口中添加 TMR3，并且注意该寄存器值的变化。

(6) 单击 Animate 并观察发生的现象。

如果一切顺利，那么将看到以下一系列现象。

(1) 主程序直接执行//4.和//5.部分的代码。

(2) 应用程序进入主循环并等待，两个定时器都在不停地计数。

(3) 定时器 2 首先到达其周期值，然后复位，并产生第一个中断（优先级 1）。

(4) 调用中断服务程序，开始执行相应的动作。

(5) 在完成//3.部分的检测后将发现情况，执行定时器 2 的中断服务程序。

(6) 接着执行一段“很长”的循环，这段时间内处理器将“卡在”此处。

(7) 定时器 3 到达其周期值，然后复位，并产生新的、具有更高优先级的中断（优先级 3）。

(8) 第一个中断服务程序被打断，并且开始执行新的中断服务程序。

(9) 程序会立刻执行到我们在定时器 3 的中断服务程序中设定的断点处（count 递增的位置），然后结束仿真。

这样，我们就能观察到中断服务程序被中断的情况。如果在这里使用动态运行功能，那么就将接着看到如下情况。

(10) 定时器 3 的中断标志被清除。

(11) 嵌套的中断服务程序结束。

(12) 返回第一个中断服务程序。

(13) 接着，将看到定时器 2 的中断服务程序结束，并返回到主循环处。

但是，请不要紧张；你可能已经注意到了，接下来并不会发生上述情况。为了使事情更“有趣”，我已经设计了一个死循环作为定时器 2 中断的中断服务程序（记作//3.）。这么做只是为了使你能有机会观看高优先级中断打断低优先级中断的现象。

只要栈还有空间并且你的思路还足够清晰，那么就可以反复嵌套多个优先级的中断。实际中，我强烈建议你不要使用两级以上的中断嵌套，否则很容易陷入极为复杂的情况，以至于难以找到出口。如果你发现自己已经进入这样的状态，那么请立刻停下来，做一个深呼吸，再重新思考。这种情况也表明你的中断优先级嵌套不够完善，或者中断服务程序太长，或者两种问题都有。通常，还有更好、更简洁的方式安排这些中断。

5.11 多重向量中断的管理

到目前为止，我们已经了解了 PIC32 单片机中断服务的基本原理。与 8 位 PIC 单片机类似，它也将所有中断源都汇集到同一个中断向量里，并指向同一个中断服务程序。这种设计极为简单，但即使考虑到 PIC32 单片机的处理速度极快（每个时钟周期可以执行一条指令），处理器依次查询所有已使能中断也会耗费大量时间，结果导致某些关键事件的响应时间可能显著延长。因此，还是有必要节省处理器响应中断的时间开销的。

为了使处理器的时间开销最小并能快速响应高优先级的中断，PIC32 单片机提供了另一种中断机制，它使用*向量中断*（vectored interrupt）和*多寄存器集*（multiple register set）。并且，PIC32MX 系列还提供 64 个向量表以及两个能够自动交换的完整寄存器集，其中每个寄存器集又包含 32 个工作寄存器。

请注意，尽管 PIC32 单片机拥有多达 96 个中断源，但是受 MIPS 内核的限制，中断向量的个数最多只有 64。所以，PIC32 的设计师将一些属于相同外围设备的中断成组地指向同一个向量（见表 5-3）。

表 5-3 PIC32MX360F512L 的向量表

向 量 号	向 量 名	注 释
0	_CORE_TIMER_VECTOR	
1	_CORE_SOFTWARE_0_VECTOR	
2	_CORE_SOFTWARE_1_VECTOR	
3	_EXTERNAL_0_VECTOR	
4	_TIMER_1_VECTOR	
5	_INPUT_CAPTURE_1_VECTOR	
6	_OUTPUT_COMPARE_1_VECTOR	
7	_EXTERNAL_1_VECTOR	
8	_TIMER_2_VECTOR	
9	_INPUT_CAPTURE_2_VECTOR	
10	_OUTPUT_COMPARE_2_VECTOR	
11	_EXTERNAL_2_VECTOR	
12	_TIMER_3_VECTOR	
13	_INPUT_CAPTURE_3_VECTOR	
14	_OUTPUT_COMPARE_3_VECTOR	
15	_EXTERNAL_3_VECTOR	
16	_TIMER_4_VECTOR	
17	_INPUT_CAPTURE_4_VECTOR	
18	_OUTPUT_COMPARE_4_VECTOR	
19	_EXTERNAL_4_VECTOR	
20	_TIMER_5_VECTOR	
21	_INPUT_CAPTURE_5_VECTOR	
22	_OUTPUT_COMPARE_5_VECTOR	
23	_SPI1_VECTOR	包括 3 个 SPI 1 中断
24	_UART1_VECTOR	包括 3 个 UART1 中断
25	_I2C1_VECTOR	包括所有的 I2C1 中断
26	_CHANGE_NOTICE_VECTOR	
27	_ADC_VECTOR	
28	_PMP_VECTOR	
29	_COMPARATOR_1_VECTOR	
30	_COMPARATOR_2_VECTOR	
31	_SPI2_VECTOR	包括 3 个 SPI 2 中断
32	_UART2_VECTOR	包括 3 个 UART2 中断
33	_I2C2_VECTOR	包括所有的 I2C2 中断
34	_FAIL_SAFE_MONITOR_VECTOR	
35	_RTCC_VECTOR	

（续）

向量号	向量名	注释
36	_DMA0_VECTOR	
37	_DMA1_VECTOR	
38	_DMA2_VECTOR	
39	_DMA3_VECTOR	
…		
44	_FCE_VECTOR	

为每组中断源分配一个独立的向量（指向独立的中断服务程序）就能省去连续测试每个中断源以确定被响应中断的时间。为了进一步缩短响应时间，很有必要使用交替的寄存器集。在中断服务程序的入口处，PIC32 单片机能够轻松地将整个工作寄存器集更换为“崭新”的，而不必将它们全部保存到栈中（标准情况下就是这么做的）。

此外，当一个或多个低优先级中断需要给更高级的中断让路时，嵌套的中断向量仍然是提供系统响应能力的有效方法。但是，由于只有一套交替的寄存器集（通常被称为影子寄存器），因此交换两次很危险。为了防止出现这种情况，寄存器集“交换”过程是自动完成的，并且只允许最高级的中断源（ipl7）使用。

只需稍做改动，我们就可以将前面的示例变成采用多重向量中断模式。

(1) 将单个中断服务程序分割为两个独立的函数。

(2) 在__ISR 宏定义中，将唯一的默认 vector 0 替换为每个中断源对应的中断向量号（参考表 5-3）。

(3) 删除中断标准测试指令。现在，毫无疑问，仅当对应的中断源出现中断标志时，才会调用每个中断服务程序。

(4) 将定时器 3 的中断优先级设定为 7，以便使用交替寄存器集功能。请记住指定的中断优先级要与__ISR() 中声明的一致。

(5) 将初始化函数调用替换为新的多重向量形式：

```
INTEnableSystemMultiVectoredInt();
```

(6) 如果你能一次就准确无误地输入前面的函数调用名，那么请发电子邮件告诉我。PIC32 函数库设计小组专门设置了个大奖，你将获得“首次输入最长函数名就拼写无误”大奖赛的冠军。

请将下面的代码保存为 multiple.c 文件，并用它作为工程的主文件。

```
/*
** Multiple Vector Interrupt
*/
#include <p32xxxx.h>
#include <plib.h>

int count;

void __ISR( _TIMER_3_VECTOR, ipl7) T3InterruptHandler( void)
{
  // 1. T3 handler is responsible for incrementing count
  count++;
```

```
    // 2. clear the flag and exit
    mT3ClearIntFlag();
} // T3 Interrupt Handler

void __ISR( _TIMER_2_VECTOR, ipl1) T2InterruptHandler(void)
{
    // 3. re-enable interrupts immediately (nesting)
    asm("ei");

    // 4. T2 handler code here
    while( 1);

    // 5. clear the flag and exit
    mT2ClearIntFlag();
} // T2 Interrupt Handler

main()
{
     //6. init timers
    PR3 = 20;
    PR2 = 15;
    T3CON = 0x8030;
    T2CON = 0x8030;

     //7. init interrupts
    mT2SetIntPriority( 1);
    mT3SetIntPriority( 7);
    INTEnableSystemMultiVectoredInt();
    mT2IntEnable( 1);
    mT3IntEnable( 1);

      //8.main loop
    while( 1);
} // main
```

像前面的示例一样生成并动态地运行该工程，就能验证程序的功能和以前相同。

首先发生定时器2中断，然后使处理器长时间保持忙碌。而定时器3中断又打断定时器2的中断服务程序，并且更新count变量的值。在这两种情况下，程序将立即并且十分有效地转移到正确的子程序中（如果我们使用的中断向量也正确）。由于定时器3的中断服务程序的序言比定时器2的更短，因此它的中断响应速度也比定时器2快（比前面的任何一个示例都快），但是这一点不容易看出来。如果希望证明这一点，可以切换到反汇编窗口并且直接比较这两个中断服务程序的序言。你就会发现定时器3的中断服务程序的序言所需的指令数仅为定时器2的一半，因此时间开销也减半。此外，由于在实际应用中主程序会变得更加复杂，所需保存的寄存器也会更多，因此这种差距还可能进一步加大。



注解 即使我们使用了交替寄存器集功能，仍然有必要缩短序言。事实上，当我们带着崭新的寄存器集进入高优先级（ipl7）中断服务程序时，至少需要初始化栈指针（它也是寄存器集的一部分），并将它从之前的寄存器集中复制出来。还需要修改PIC32的中断优先级屏蔽字（IM），以便关闭低优先级的中断。即使这样，在最快的情况下，序言仍需要执行7条汇编指令。

5.12 一个简单的应用示例

再增加一些程序，就能将前面的示例转变成一个更实用的应用程序。其中，定时器1用于

维持实时时钟跟踪十分之一秒、数秒以及数分钟。为了产生一个简单的形象化的反馈，我们将使用 PortA 的低 8 位作为二进制显示，以表示程序的运行秒数。下面是具体的实施过程。

- 声明一些新的整型变量，作为秒和分的计数器：

```
int dSec = 0;
int Sec = 0;
int Min = 0;
```

- 在中断服务程序中递增十分之一秒计数器：

```
dSec++;
```

注意：为简单起见，本章将假设 PIC32 采用 16MHz 的系统和外围设备时钟。在第 7 章，我们将介绍一些有关振荡器的详细内容，并将学习如何使单片机运行在更高的时钟频率下，从而使器件的性能最优。

为了实现秒和分钟的进位，需要增加以下代码。

- 将定时器 1 的预分频系数设位 1:64，以实现所需的定时周期。

```
T1CON=0x8020;
```

- 配置定时器 1 的周期寄存器（假设外围设备时钟为 16MHz，对应的周期为 62.5ns）以产生 1/10 秒中断：

```
PR1=25000-1; // 25,000 * 64 * 62.5ns=0.1 s
```

- 将 PortA（LSB）配置为输出方式，关闭 JTAG 端口以便完全控制所有的 LED，相关代码为：

```
DDPCONbits.JTAGEN = 0;
TRISA = 0xff00;
```

- 在主循环中增加代码，不断用秒计数器的当前内容刷新 PortA（LSB）：

```
PORTA = Sec;
```

将上述新代码保存为 clock.c，并且将它作为工程的主文件。下面是完成后的代码：

```
/*
** A real time clock
**
** example 5
*/

#include <p32xxxx.h>
#include <plib.h>

int dSec = 0;
int Sec = 0;
int Min = 0;

// 1. Timer1 interrupt service routine
void __ISR( 0, ipl1) T1Interrupt( void)
{
    // 1.1 increment the tens of a second counter
    dSec++;

    if ( dSec > 9)       // 10 tens in a second
    {
      dSec = 0;
      Sec++;                    // increment the seconds counter
      if ( Sec > 59)    // 60 seconds make a minute
      {
```

```
            Sec = 0;
            Min++;    // increment the minute counter

            if ( Min > 59)  // 59 minutes in an hour
                Min = 0;
          } // minutes
        } // seconds

    // 1.2 clear the interrupt flag
    mT1ClearIntFlag();
  } //T1Interrupt

main()
{
    // 2.1 init I/Os
    DDPCONbits.JTAGEN = 0;       // disable JTAG port
    TRISA=0xff00;                // set PORTA LSB as output

    // 2.2 configure Timer1 module
    PR1 = 25000-1;  // set the period register
    T1CON = 0x8020; // enabled, prescaler 1:64, internal clock

    // 2.3 init interrupts
    mT1SetIntPriority( 1);
    mT1ClearIntFlag();
    INTEnableSystemSingleVectoredInt();
    mT1IntEnable( 1);

    // 2.4. main loop
    while( 1)
    {
      // your main code here
      PORTA=Sec;
    } // main loop
} // main
```

为了用 MPLAB SIM 仿真器测试新工程，请完成以下步骤。

(1) 打开 Watch 窗口（请将它固定在你喜欢的位置）。

(2) 添加以下变量：

- ❑ dSec，在 Symbol 下拉选项框中选择，然后单击 Add Symbol 按钮；
- ❑ TMR1，在 SFR 下拉选项框中选择，然后单击 Add SFR 按钮；
- ❑ Status，在 SFR 下拉选项框中选择，然后单击 Add SFR 按钮。

(3) 打开仿真器 StopWatch 窗口（选择菜单 Debugger|StopWatch）。

(4) 在中断服务程序的 1.1 部分后添加一个断点。将光标指向该行，并选择 Set Breakpoint 选项，或者直接双击该行。在这里设置断点后，就能观察到中断是否被触发。

(5) 执行运行命令（选择菜单 Debugger|Run 或者按 F9 键）。仿真器将很快停止，光标位于中断服务程序中的断点处。

这样，就真的停在中断服务程序内部了！这意味着触发事件已被激活，也就是说，定时器 1 的计数值达到了 24 999（请注意，定时器是从 0 开始计数的，因此到 24 999 正好数了 25 000 次），它再乘以预分频系数，也就是 25 000 × 64，正好是经过了 160 万个时钟周期。

StopWatch 窗口可以确定到目前为止的总周期数，事实上，它比 160 万略大。StopWatch 计数器还包括了程序初始化所需的时间。以 PIC32 单片机的执行速度（每秒 1600 万条指令）来计算，恰好是 1/10 秒！

从 Watch 窗口中，我们可以查看处理器中断优先级屏蔽位（IM）的当前值，它位于 Status 寄存器中。由于我们已经进入优先级为 ipl1 的中断服务程序中，因此将看到状态寄存器第 10～15 位的值等于 1。

在图 5-2 中，我已经在 Watch 窗口中圈出了 Status 寄存器中包含中断屏蔽字（IM）的部分。此外，StopWatch 窗口显示了从开始到第一个断点所消耗的时间（单位是微秒）。从当前位置开始单步运行（使用 StepOver 或者 StepIn 命令）就可以监视中断服务程序中下一条指令的执行情况。当中断服务程序执行完毕后，就会看到中断屏蔽字恢复为零。

（1）再次执行 Run 命令后，将看到程序计数器（由绿色箭头表示）会再次指向中断服务程序。这次，将看到消耗时间增加了准确的 160 万个时钟周期。

（2）在 Watch 窗口中添加 Sec 和 Min 变量。

（3）多执行几次 Run 命令，待执行 10 次后，就会发现秒计数器 sec 增加 1。

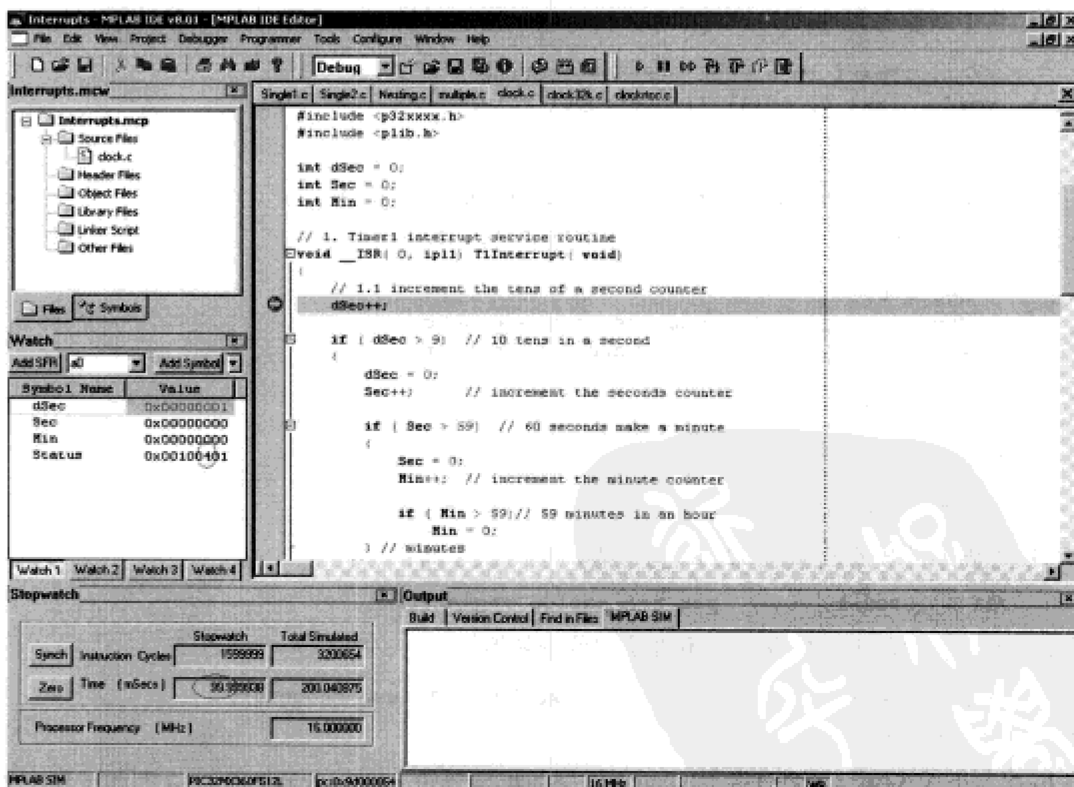


图 5-2 Clock.c 程序仿真时的屏幕截图

为了测试分钟计数器的递增，就要删除当前的断点，并且在下面几行的位置放置一个新的断点；否则，你必须执行 Run 命令 600 次才能看到分钟计数器加 1！

（1）在代码的 1.2 部分的 Min++指令处放置一个新的断点。

（2）执行 Run 命令，可以看到秒计数器已经被清零。

（3）执行 Step Over 命令，可以看到分钟计数器加 1。

中断服务程序每 1/10 秒执行一次，至此已经执行了 600 次。同时，主循环中的代码则在连续执行，并耗费了 9600 亿个时钟周期。说实话，我们的示例程序并没有利用所有的时钟周期，

而是浪费在更新 PortA 的内容上。在实际的应用系统中，要完成很多任务，并且始终都有精确的实时时钟负责计数。

5.13 辅助振荡器

PIC32 单片机的定时器 1 模块的另一项功能（之前的 8 位和 16 位 PIC 单片机也有）是作为实时时钟。事实上，定时器 1 除了可以使用高速时钟源之外，还可以接一个低频振荡器（即所谓的辅助振荡器）。由于定时器 1 可以在低频工作（通常是与低廉的 32 768Hz 晶振相接），因此功耗很低。此外，由于该辅助振荡器与主时钟电路独立，因此当主时钟关闭并且处理器进入低功耗模式时，它仍然能够工作。事实上，辅助振荡器是系统工作在低功耗模式的重要部分。有时它还被用于替代主时钟，其他情况下则供定时器 1 或选定的一组外围设备使用。

为了使前面的示例程序转为使用辅助振荡器，我们需要做以下改动。

- 将中断服务程序改为只对秒和分钟计数，慢速时钟不需要再对 1/10 秒计数：

```
// 1. Timer1 interrupt service routine
void __ISR( 0, ipl1) T1Interrupt( void)
{
  // 1.1
  Sec++;     // increment the seconds counter
  if ( Sec > 59)   // 60 seconds make a minute
  {
  Sec = 0;
  Min++;     // increment the minute counter
  if ( Min > 59)   // 59 minutes in an hour
      Min = 0;
  } // minutes

  // 1.2 clear the interrupt flag
  mT1ClearIntFlag();
} //T1Interrupt
```

- 将周期计数器改为每 32 768 个周期产生一次中断：

```
PR1 = 32768-1;     // set the period register
```

- 改变定时器 1 的配置字（不再使用预分频器）

```
T1CON = 0x8002;    // enabled, prescaler 1:1, use secondary
oscillator
```

但是，由于 MPLAB SIM 无法全面支持辅助振荡器输入，因此我们无法立即测试新配置。

在后面的课程里，我们将学习如何使用新工具产生激励文件，用它可以方便地测试 32kHz 晶振连在 PIC32 单片机 T1CK 和 SOSCI 引脚上的情况。

5.14 实时时钟和日历（RTCC）

在生成前面两个工程时，我们已经学会逐步使用实时时钟实现日历，只要再加上日、周、月和年即可实现一个包括完整功能的日历。

这几行代码可能每天、每月甚至每年才执行一次，因此对整个系统的性能没什么影响。尽管偶尔开发这样的代码也很有趣，比如考虑到闰年并能制定出所有的细节等，但是 PIC32MX 系列单片机已经自带了完整的实时时钟和日历模块（RTCC），可供随时使用。

多么方便啊！它不仅使用到同一个低频辅助振荡器，而且还带有振铃和笛音，此外还有一个能够产生中断的闹钟功能。换句话说，该模块一旦被初始化，就能激活 RTCC 模块，并且等着产生中断了。例如，该中断可以设定在一年的某月某日某时某分某秒产生（如果你设定的是 2 月 29 日，那么就是 4 年产生 1 次）。

下面是 RTCC 中断服务程序的示例：

```
// 1. RTCC interrupt service routine
void __ISR( 0, ipl1) RTCCInterrupt( void)
{
  // 1.1 your code here, will be executed only once a year
  // or once every 365 x 24 x 60 x 60 x 16,000,000 MCU cycles
  // that is once every 504,576,000,000,000 MCU cycles

  // 1.2 clear the interrupt flag
  mRTCCClearIntFlag();
} // RTCCInterrupt
```

为了初始化 RTCC 模块，我们还需要修改主程序。只有以正确的顺序访问很多寄存器并填写正确的数据，才能将 RTCC 配置好。幸运的是，RTCC 配置函数已成为标准 PIC32 外围设备库函数的一部分，因此我们可以使用这套强大的函数集轻松地完成对 RTCC 的配置。下面就是与初始化相关的代码：

```
main()
{
  // 2.1 init I/Os
  DDPCONbits.JTAGEN = 0;       // disable JTAG port
  TRISA = 0xff00;              // set PORTA LSB as output

  // 2.2 configure RTCC module
  RtccInit();                  // inits the RTCC
  // set present time
  rtccTime tm; tm.sec=0x15;tm.min=0x30;tm.hour=01;
  // set present date
  rtccDate dt;
  dt.wday=0; dt.mday=0x15; dt.mon=0x10; dt.year=0x07;
  RtccSetTimeDate(tm.l, dt.l);

  // set desired alarm to Feb 29th
  dt.wday=0; dt.mday=0x29; dt.mon=0x2;
  RtccSetAlarmTimeDate(tm.l, dt.l);

  //2.3 init interrupts,
  mRTCCSetIntPriority( 1);
  mRTCCClearIntFlag();
  INTEnableSystemSingleVectoredInt();
  mRTCCIntEnable( 1);

  //2.4 main loop
  while( 1)
  {
    // your main code here
    // . . .
  } // main loop
} // main
```

5.15 小结

在本章中，我们学习了如何轻松地编写中断服务程序代码。这要多谢 MPLAB C32 编译器带来的 C 语言扩展功能以及 PIC32 单片机强大的中断控制功能。中断是帮助嵌入式控制系统程序员管理多任务的极为有效的工具，它能满足精确的时间和资源约束要求。同时，它又是一个极强大的故障源。在 PIC32 单片机参考手册和 *MPLAB C32 User Guide*（MPLAB C32 用户指南）中，你能找到比本章介绍的更多的有用信息。本章还介绍了更多有关使用定时器 1 和辅助低功耗振荡器的知识，并且概述了实时时钟和日历模块（RTCC）的功能。

5.16 对 PIC 单片机行家的提示

请注意，PIC32 单片机有一对方便的指令，它们可以一次打开或关闭所有的中断。如果有一部分代码需要暂时关闭所有的中断，那么就可以使用如下的汇编指令：

```
asm("di");
. . .                    // protected code here
asm("ei");
```

但是，如果你想保护一段代码不受中断影响，而不知道中断是否已经打开/关闭，那么就可以采用更为谨慎的方法——调用 plib.h 函数库里的两个函数：

- ❑ `INTDisableInterrupts();`它不仅能关闭中断，还能返回对应于原中断状态的值；
- ❑ 当操作结束时，请使用 `INTRestoreInterrupts(status);` 函数恢复系统原先的状态。

5.17 提示与技巧

根据 PIC32 的数据手册，为了激活辅助低功耗振荡器，需要将 OSCCON 寄存器里的 SOSCEN 位置位。但是当你匆忙地在最后一个示例中输入代码并打算在实际的目标板上执行前，请注意 OSCCON 寄存器，由于它包含对单片机的重要控制，影响到主振荡器及其速度的选择，因此受到锁定保护。为了安全测试，必须首先按照特定的顺序完成解锁，否则你的命令将被忽略。此时，PIC32MX 的外围设备函数库帮了我们的忙，它包含大量有用的函数，能够完成振荡器模块的配置以及所有必需的加锁和解锁操作，具体如下所示。

- ❑ `mOSCEnableSOSC()`，它能在运行时打开或关闭（`mOSCDisableSOSC()`）外部辅助振荡器（SOSC）。
- ❑ `OSCConfig()`，它能动态（在程序运行时）改变期望的时钟源、PLL 倍频系数、PLL 预分频系数以及 FRC 分频系数。
- ❑ `mOSCSetPBDIV()`，它能动态改变外围设备总线时钟的分频系数。使用该函数时必须特别小心，因为它还将同时影响到所有外围设备的运行。



注解 只有在时钟切换配置位被启用时，改变时钟源才能成功。请检查菜单 Configure|Configure bits 或者配置位语句#pragmas 的设置。

此外，当需要将 PIC32MX 单片机配置为工作在空闲（IDLE）和休眠（SLEEP）模式时，需要注意下面两个函数。

- mPowerSaveSleep()函数。它将关闭 PIC32 单片机的系统时钟和外围设备总线时钟，并使器件进入超低功耗工作方式。任何复位和活动的异步（请记住外围设备时钟是关闭的）外围设备事件都会唤醒器件，即使对应的中断未打开也能被唤醒。有效的唤醒源包括电平变化通知输入、外部中断输入、复位以及掉电复位（Brown Out）信号等。
- mPowerSaveIdle()函数。它将关闭系统时钟，但是又能保持外围设备时钟继续运行。任何主动的外围设备中断源都可以唤醒器件。例如，有效的唤醒源有 UART、SPI、定时器、输入捕获器、输出比较器以及大部分其他外围设备。

5.18 练习

为以下外围设备编写中断服务程序：

（1）边沿可选择中断；

（2）电平变化通知中断；

（3）输出比较。

5.19 参考书

Keith E. Curtis 所著的 *Embedded Multitasking*。本书的作者很熟悉多任务系统，并且在书中建立了很多小型而高效的嵌入式控制应用系统。

5.20 链接

http://en.wikipedia.org/wiki/Interrupts。这是一个很好的中断入门网站。

http://en.wikipedia.org/wiki/Computer_multitasking。这是一个可以继续学习多任务、特别是了解实时多任务和处理异步事件方面知识的网站。

第 6 章　存　储　器

6.1　计划

使用高度集成的单芯片微处理器的好处在于，设备的尺寸减小了、鲁棒性增强了，并且预配有成套且可立即使用的外围设备。然而，大多数嵌入式控制系统设计者很快就会意识到，器件上可用的存储器容量（Flash 和 RAM）将对产品的成本和可用性产生最直接的影响。因此有必要学习如何充分利用这些存储器。

在本章中，我们将回顾 C 语言中有关字符串型数据的声明和操作的基本知识，并以此为练习来研究 MPLAB C32 编译器所使用的存储器分配技术。PIC32 内核具备一些 8 位或 16 位 PIC 单片机所没有的先进功能，包括存储器空间重映射、缓冲存储器的内容以及与 DMA（Direct Memory Access，直接存储器访问）控制器共享存储器总线等功能。为了研究 MPLAB C32 编译器和链接器是如何联合工作以产生最紧凑且高效的代码的，我们将使用反汇编列表窗口、存储器窗口以及映射文件等工具。

6.2　准备

本章只使用软件工具，包括 MPLAB IDE、MPLAB C32 编译器以及 MPLAB SIM 仿真器。

请使用 New Project Setup（创建新工程）检查表创建名为 Strings 的工程以及名为 strings.c 的源文件。

6.3　探索

C 语言将字符串视为 ASCII 字符数组。字符串中的每个字符都以连续的 8 位整型数组的形式依次存储在存储器中。在字符串最后一个字符的后面，增加了一个值为 0 的字节（用字符'\0'表示）作为终止符。



注解　这只是适用于标准 C 字符串操作函数库 string.h 的一个规则。当然，也可以定义一个完全不同的函数库，比如在存放字符串的数组的第一个元素内存放数组的长度。事实上，Pascal 程序员对这种定义方式很熟悉。

下面将首先回顾包含一个字符的变量的声明方法：

```
char c;
```

和前面几课讲到的一样，我们就是这么定义 8 位整数（字符）的，它默认是有符号数（表示范围是−128～127）。

还可以在声明时对其赋数值：

```
char c = 0x41
```

也可以在声明时对其赋 ASCII 码数据：

```
char c = 'a'
```

注意，在赋值 ASCII 字符时要使用单引号。你可能发现，上述两种赋值的结果相同，这是

因为对于 C 编译器来说，这两种声明方式没有区别，字符就是数字。

下面将以 8 位整数（字符）的形式声明并初始化一个字符串：

```
char s[5] = { 'H', 'E', 'L', 'L', 'O'}
```

本例采用标准的数值型数组的记法来初始化数组。也可以采用为初始化字符串而特别创立的更加方便的记法（简化方式）：

```
char s[5] = "HELLO";
```

为了进一步简化，以便不必计算字符串包含的字符个数（也是为了防止人们出错），还可以采用下面的记法：

```
char s[] = "HELLO";
```

MPLAB C32 编译器将自动确定需要存储在字符串中的字符个数，并且自动添加终止符（零）。终止符在后面的字符串操作子程序里确定字符串的长度时十分有用。因此，上面的实例事实上相当于：

```
char s[6] = { 'H', 'E', 'L', 'L', 'O', '\0' };
```

对 char 型（8 位整数）变量赋值，并对其进行算术操作，这与对其他类型的整数进行相同操作没有区别：

```
char c;         // declare c as an 8-bit signed integer
c = 'a';        // assign the value 'a' from the ASCII table
c ++;           // increment it. . .
                // it will represent the ASCII character 'b'
```

对字符数组（字符串）的任何元素也可以进行类似的操作。但是无法用简化方式在初始化时给整个字符串赋值：

```
char s[15];     // declare s as a string of 15 characters
s = "Hello!";   // Error! This does not work!
```

如果在源文件的顶部引用 string.h 文件，就可以访问很多有用的函数。这些函数可以进行以下操作。

- ❑ 将一个字符串的内容复制到另一个字符串，例如：

  ```
  strcpy( s, "HELLO");      // s : "HELLO"
  ```

- ❑ 合并（连接）两个字符串，例如：

  ```
  strcat( s, "WORLD");     // s : "HELLO WORLD"
  ```

- ❑ 确定字符串的长度，例如：

  ```
  i = strlen( s);           // i : 11
  ```

此外，还有很多其他功能。

6.4 存储空间的分配

编译器的任务是产生操作变量的代码，而将变量放在存储器的什么位置则是由链接器负责确定的，它会为每个对象在可用存储器空间中寻找一个物理地址。和数值的初始化一样，字符串的声明和初始化也是如此：

```
char s[] = "Exploring the PIC32";
```

这样就会发生 3 件事情。

- ❑ MPLAB C32 链接器将保留一段连续的存储器空间（在 RAM 中）来保存变量，在本例

中就是 20B。该空间属于所谓的 data 段。

- MPLAB C32 链接器将初始化数据保存在一个 20B 的表格里（位于 Flash 程序空间）。该空间属于所谓的 rodata 代码段（或者称为只读段）。
- MPLAB C32 编译器产生一小段子程序（是前面提到的启动代码的一部分），它在执行 main() 函数前被调用，可将上述数据调入 RAM，从而完成变量初始化。

换句话说，字符串“Exploring the PIC32”共使用了两次存储器空间，它的副本保存在 Flash 程序空间中，而 RAM 存储器仍然为它保留着空间。此外，必须考虑执行初始化以及实际数据复制所花费的时间。如果在程序执行时并不需要对字符串操作，而只是原样使用，比如说传输至串行端口或者发送到屏幕上去显示，那么就不必浪费宝贵的存储器资源了。可以将字符串声明为常数以节省 RAM 空间、缩短初始化代码并节省初始化时间：

```
const char s[] = "Exploring the PIC32";
```

这样，MPLAB C32 链接器就会只在程序存储器的 rodata 代码段中分配空间，此空间里的字符串可以直接被访问。编译器将把字符串当作直接指向程序存储器的指针，从而不会浪费 RAM 空间。

在本章前面的例子中，我们看到其他字符串被隐式地定义为常数；比如，我们编写如下的代码：

```
strcpy( s, "HELLO");
```

这里的字符串“HELLO”就被隐式地定义为 const char 型，并且被分配到程序存储器的 rodata 段。



注解　*如果在程序中需要多次使用同一个常数字符串，那么即使关闭了编译器的所有优化选项，MPLAB C32 编译器也会自动在 rodata 段只保存一个副本，以优化存储器的使用。*

下面，将开始使用 MPLAB SIM 仿真器及以下代码段来研究存储器的分配问题。

```
/*
** Strings
*/
#include <p32xxxx.h>
#include <string.h>

// 1. variable declarations
const char a[] = "Exploring the PIC32";
char b[100] = "Initialized";

// 2. main program
main()
{
  strcpy( b, "MPLAB C32");     // assign new content to b
} // main
```

(1) 利用 Project Build（生成工程）检查表生成该工程。

(2) 打开 Watch（观察）窗口（并将它固定在自己喜欢的位置）。

(3) 从符号选择框中选择变量 a 和 b，然后单击 Add Symbol 按钮将它们添加至 Watch 窗口（参见图 6-1）。

窗口中的小+号表示该变量是数组，可以将其展开以便观察数组中的每个元素（参见图 6-2）。

默认情况下，MPLAB 将以十六进制格式显示数组的每个元素，但是用户可以将其改为显示 ASCII 字符或者个人喜欢的其他形式。

（1）选择数组的一个元素。

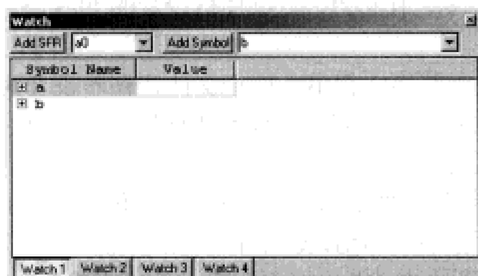


图 6-1 包含两个字符串的 Watch 窗口

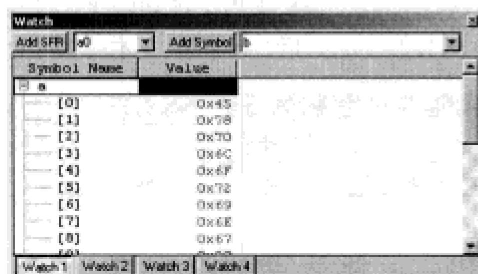


图 6-2 字符串展开后的视图

（2）单击鼠标右键弹出 Watch（观察）窗口菜单。

（3）选择 Properties（属性）选项（菜单的最后一项）。

这样就可以看到 Watch 窗口的 Properties 对话框（参见图 6-3）。

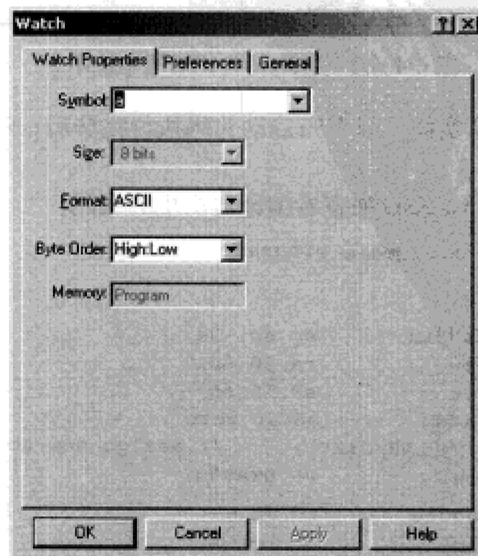


图 6-3 Watch 窗口的属性对话框

在该对话框中，用户可以改变所选数组元素的数据显示格式，还可以看到 Memory（存储器）场（灰色显示），它将告诉用户所选变量是位于数据空间还是程序空间。

如果选择了常数字符串 *a* 的 Properties 对话框，就会看到它的存储器空间是程序空间（Program），可见常数字符串只使用了 PIC32 单片机少量的 Flash 存储器空间，并且不需要为它分配 RAM 空间。

相反，Properties 对话框则给出字符串 *b* 位于文件寄存器或者其他 RAM 空间。

此外，通过在 Watch 窗口中添加新列，就能同时以多种格式显示同一个变量的值，具体方法如下。

（1）选择 Watch 窗口中表格区的最顶行（Value 列右侧的位置）。

（2）选择任何其他格式（比如，选择 Char）。

（3）只要窗口内还有空间，就可以根据需要重复上述步骤。

下面将研究字符串 *a* 是如何被初始化的。从 Watch 窗口来看，在工程生成后，它就可以使用了。

而另一方面，字符串 *b* 仍然是空的，看起来尚未被初始化。只有当启动 MPLAB SIM 仿真器并且首次单击复位按钮到达主函数的开始处后，字符串 *b* 才被初始化为正确的值（参见图 6-4）。

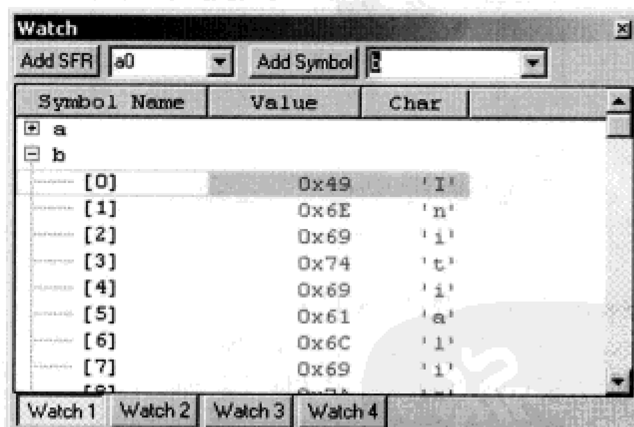


图 6-4　执行完启动代码后的字符串 *b*

可见，变量 *b* 位于 RAM 空间。只有当启动代码执行完毕后，这些变量才能完成初始化并且可用。

我们将再次使用反汇编列表窗口来查看编译器产生的代码：

```
14:                    // 2. main program
15:main()
16:{
9D000018    27BDFFE8    addiu       sp,sp,-24
9D00001C    AFBF0014    sw          ra,20(sp)
9D000020    AFBE0010    sw          s8,16(sp)
9D000024    03A0F021    addu        s8,sp,zero
17:         strcpy( b, "MPLAB C32");      // assign new content to b
9D000028    3C02A000    lui         v0,0xa000
9D00002C    24440000    addiu       a0,v0,0
9D000030    3C029D00    lui         v0,0x9d00
9D000034    2445074C    addiu       a1,v0,1868
q9D000038   0F400016    jal         0x9d000058
9D00003C    00000000    nop
```

```
18:                      } // main
9D000040    03C0E821    addu       sp,s8,zero
9D000044    8FBF0014    lw         ra,20(sp)
9D000048    8FBE0010    lw         s8,16(sp)
9D00004C    27BD0018    addiu      sp,sp,24
9D000050    03E00008    jr         ra
9D000054    00000000    nop
```

可以看到 main() 函数很短，紧接着就是位于列表底部的库函数 strcpy() 的汇编语言程序。不要因为程序的长度和表面上的复杂性而分散精力；这是一段优化得很好的代码，它充分利用了 PIC32 单片机的 32 位总线以及高速缓存。然而，有关它的分析已经超出了本章的研究范围。

尽管 string.h 库包含很多函数，并且头文件 string.h 中包含了所有函数的声明，但是链接器做得很巧妙，它只添加了实际使用的函数。于是，我们幸运地只需添加一个子程序。

6.5 查看映射

我们需要掌握的另一个工具是.map 文件，它有助于我们理解字符串（更常见的情况是数组变量）是如何被初始化并且被分配到存储器中去的。该文本文件由 MPLAB C32 链接器产生，并能在 MPLAB 编辑器中查看，它是专门设计用于帮助程序员理解和解决存储器分配问题的。

该文件位于主工程文件夹中，那里还有工程包含的所有源文件。请选择菜单 File|Open 浏览到工程文件夹。MPLAB 编辑器会默认列出所有的.c 文件，用户可以将文件类型改为.map（参见图 6-5）。

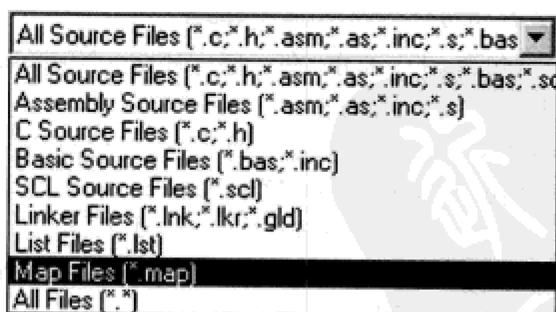


图 6-5 选择.map 文件类型

映射文件看起来很冗长，但是只要研究其中的一些关键部分，就能发现很多有用的数据。该文件主要包括 3 部分。

- 工程包含的文档列表。这是一个文件名列表，包括所有库函数、生成工程时链接器使用的目标文件、被引用的文件以及需要的特殊符号等。这些文件大部分都会被链接器脚本自动引用，但是你可以迅速找到有一行是我们的主目标文件 string.o，由于它调用了函数 strcpy()，因此导致 strcpy.o 也被链接进来。以下给出的就是文件中的这行内容。

```
C:/Program Files/Microchip/../pic32mx/lib\libc.a(strcpy.o)
                                  Strings.o  (strcpy)
```

- 存储器配置表。它包含工程使用的每个存储区（包括数据存储区和程序存储区）的位

置和大小，并要适合所选择的 PIC32 器件的配置。以下给出该表的内容：

```
Memory Configuration

Name                         Origin             Length
Attributes
kseg0_program_mem      0x9d000000      0x00080000      xr
kseg0_boot_mem         0x9fc00490      0x00000970
exception_mem0         x9fc01000       0x00001000
kseg1_boot_mem         0xbfc00000      0x00000490
debug_exec_mem         0xbfc02000      0x00000ff0
config3                0xbfc02ff0      0x00000004
config2                0xbfc02ff4      0x00000004
config1                0xbfc02ff8      0x00000004
config0                0xbfc02ffc      0x00000004
kseg1_data_mem         0xa0000000      0x00008000      w !x
sfrs                   0xbf800000      0x00100000
*default*              0x00000000      0xffffffff
```

你会发现，有些存储区域的名称很直观易懂，而有些看起来则十分令人费解（这部分都沿袭了 MIPS 的悠久传统）。

- 链接器脚本和存储器映射。这是文件中最长的一部分，包含看起来无休止的存储器段名称的列表。根据链接器脚本的严格规则，每个存储器段最终都要被链接器放入前面列出的某个存储区内。在这部分中，我们最感兴趣的是以下内容。

(1) .reset 段，包含链接器放置的复位向量。它通常被填写为默认的函数（_reset()）：

```
.reset       0xbfc00000      0x10 C:/. . ./pic32mx/lib/crt0.o
                0xbfc00000             _reset
```

(2) .vector_x 段，共包含 64 个部分，每一部分对应一个中断服务程序。除非程序使用特殊的中断服务程序，否则不为空。

```
.vector_0      0x9fc01200      0x0
```

(3) .startup 段，C0 初始化代码就放在这里。

```
.startup       0x9fc00490      0x1e0 C:/. . ./lib/crt0.o
```

(4) .text 段，包含很多内容，MPLAB C32 编译器根据源文件所产生的所有代码都放在这里。下面这部分是由 main() 函数产生的：

```
.text          0x9d000018      0x40  Strings.o
             0x9d000018               main
```



注解 尽管有时候段名称（比如.text）会引起歧义，但是它们都遵从了 C 编译器悠久的历史规范，从第一个 C 编译器起一直沿用至今。

(5) .rodata 段，位于程序存储器空间，用于放置只读（常数）数据。在这里，将找到常数字符串 *a*，比如：

```
.rodata        0x9d000738      0x20Strings.o
             0x9d000738               a
```

(6) .data 段，全局变量放置于此，它们都被分配到 RAM 存储器中：

```
.data          0xa0000000      0x64  Strings.o
             0xa0000000               b
```

（7）最后是指向.data1段的一个指针，需要利用C0代码载入到变量b的初始值就放在这里，它也位于程序存储器空间：

```
*(.data1)
               0x9d00076c        _data_image_begin=LOADADDR(data)
```

为了验证这些地址单元到底保存着什么数据，需要使用Memory窗口（选择View|Memory）。然后选择Data View（数据视图）选项卡，就可以以经典的十六进制堆（hex dump）格式查看存储器的内容。在存储器窗口单击，并在弹出的上下文菜单中选择Go To（或者按Ctrl+G组合键）就可以激活Go To（跳转）对话框（参见图6-6）。

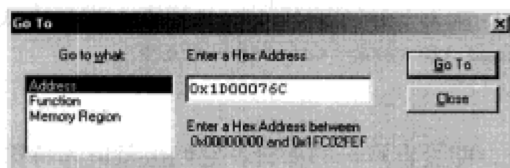


图6-6 Memory窗口的Go To对话框

在十六进制地址框内，输入在前面找到的地址（0x9d0076c），然后单击Go To按钮。Memory窗口会自动居中所选的地址单元，这样就能看到我们正在寻找的初始化值。

```
Address       00        04        08        0C          ASCII

1D00_0760  9D0003AC  9D0004F4  9D000578  74696E49  ........ x...Init
1D00_0770  696C6169  0064657A  00000000  00000000   ialized. ........
```

6.6 指针

指针是间接引用（指向）其他变量或者其他变量部分内容的变量。在C语言编程中，指针和字符串是分不开的，它们通常是处理所有数组数据类型的有力工具。事实上，它们的功能太强大了，以至于它们也是程序员手上最危险的工具之一和主要的程序缺陷源。有些编程语言（比如Java）已经完全禁止使用指针，以便提高语言的鲁棒性和可验证性。

MPLAB C32编译器利用PIC32架构能轻松地管理大容量的数据存储器和程序存储器（高达4GB）。MPLAB C32编译器不区分指向数据存储器对象的指针和位于程序存储器空间的const对象。因此只用一套标准函数，就能根据需要来操作数据和程序存储空间的变量和一般的存储器块。

下面的经典程序示例比较了使用指针和索引实现对整数数组的连续访问的差别：

```
int   *pi;       // define a pointer to an integer
int    i;        // index/counter
int a[10];       // the array of integers

// 1. sequential access using array indexing
for( i=0; i<10; i++)
    a[ i]= i;

// 2. sequential access using a pointer
pi=a;
for( i=0; i<10; i++)
{
    *pi= i;
    pi++;
}
```

第 1.部分使用了简单的 for 循环，并且每轮循环中都使用 *i* 作为数组的索引。为了赋值，编译器将 *i* 值乘以矩阵元素的字节数（4），然后将结果作为偏移量加在数组 *a* 的初始地址上。

第 2.部分将指针初始化为指向数组 *a* 的初始地址。在每轮循环中都只是使用指针间接操作符（*）实现赋值操作，然后再将指针加 1。

对比这两种方法，可以看出，使用指针能够在每轮循环中节省至少一次乘法运算。如果在循环中多次使用数组元素，那么性能改善就会成倍增强。

指针的语法在 C 语言中非常“简明”，它能实现一些极为高效的代码，但同时又增加了出错的机会。

你至少应当熟悉最常见的缩写。前面的代码段还能进一步简化成如下形式：

```
// 2. sequential access to array using pointers
for( i=0, p=a; i<10; i++)
  *pi++ = i;
```

此外，还要注意空指针，即没有目标的指针，它们被赋值为特殊值 NULL，stddef.h 库中有它明确的实现和定义。

6.7 堆

使用指针的优点之一是，能够操作在存储器中动态（即在运行时）定义的对象。堆是数据存储器中被保留用作这种功能的一块区域，而标准 C 函数库 stdlib.h 的部分函数则可作为分配和释放存储器块的工具。这类函数中至少包含这两个基本函数：malloc()和 free()。

```
void *malloc(size_t size);
```

该函数从堆中获得期望大小的一块存储器空间，并且返回一个指向它的指针。

```
void free(void *ptr);
```

该函数则将指针 ptr 指向的存储区归还给堆。

MPLAB C32 链接器将堆放置在工程所有全局变量之前的空闲 RAM 存储器空间以及保留的栈空间内。尽管链接器知道空闲存储器的数量，但还是必须明确地指示链接器保留精确数量的空间供堆使用，其默认值等于零。

选择菜单 Project|BuildOptions|Project 打开 Build Options（生成选项）对话框，选择 MPLAB PIC32 Linker 标签，然后定义堆的大小（单位是字节）。

通用的规则是，尽可能为堆分配最大的存储器空间。这将使 malloc()函数能够最有效地利用可用的存储器。毕竟，如果不将这些可用的存储器分配给堆，那么它们也没有机会被用到。

6.8 PIC32MX 总线

如果前面几节学习 MPLAB C32 编译器和链接器分配变量的技术已经使你有些头晕，那么你现在可能想先休息一下！

但如果前述内容只是增加了你的好奇心，那么就请随我继续探索并研究 PIC32 存储器总线基本架构的设计初衷。

PIC32 单片机的架构与你以前所熟悉的 PIC 单片机（8 位和 16 位）的架构有所不同。PIC32 采用了更为传统的冯·诺依曼存储架构而不是经典的（PIC）哈佛架构。其最大变化是不再需要两条完全分离且独立的总线。现在将采用同一条（32 位）总线访问程序存储器（Flash）和数据存储器（RAM）。

冯·诺依曼架构是一种更经济的实现方式（两套独立的 32 位总线非常昂贵），同时又提供

一套更为简单的统一编程模型，从而不再需要8位和16位哈佛架构使用的很多“技巧”（比如访问程序存储器数据表），并最终消除了二者的壁垒，从而首次使PIC处理器能够在RAM存储器中执行代码！

没有了以前的一些优点，看起来将导致芯片的性能下降，然而事实并非如此。事实上，PIC32单片机采用的五级流水线结构和预取指令结构使它能够史无前例地在保持每个时钟周期一条指令的处理速度的同时，又能高效地访问总线。



注解 在下一章，我们将有机会详细了解存储器缓存单元的运行情况，并且分析它对器件性能的影响。在这里先不多说了，我只想指出一条重要细节。PIC32的内核和cache模块实际上是由名为I和D的两条独立的32位总线相连。它们使处理器能够同时从cache中取指令和数据。因此，PIC32到底是哈佛机还是冯·诺依曼机呢？这将留给你自己考虑。对我来说最要紧的是它运行得真快、真高效！

在相同的时钟频率（比如说20MHz）下，PIC32单片机每秒执行的指令数是PIC16或者PIC18单片机的4倍。也就是说，PIC32单片机每秒能够执行2000万条指令，而PIC16或者PIC18单片机每秒只能执行500万条指令。同时也意味着，在相同的时钟频率下，PIC32单片机每秒执行的指令数是PIC24单片机（比如dsPIC30或者dsPIC33）的2倍。如果再考虑到现在每条PIC32指令能够直接处理32位宽的整数（而不是8位或者6位），那么就可以感受到PIC32运算能力有效提高了。

在下一章，我们将进一步研究PIC32单片机的振荡器和时钟管理电路的运行情况。我们还将介绍更多有关预取指令和数据cache运行方面的细节，这将有助于我们理解PIC32架构面临的新的性能限制，以及我们该如何配置器件以使它达到最优的性能和功耗等级。

6.9 PIC32MX 存储器映射

PIC32的核心是MIPS内核，它具有很多先进特性，比如可以通过使用MMU（memory management unit，存储器管理单元）将应用程序所用的存储器空间与操作系统所用的存储器空间分离开来，并且还有两种不同的运行模式：*用户模式*和*内核模式*。由于PIC32MX系列器件明显是面向嵌入式控制应用系统的，而这些应用往往不需要如此复杂，因此PIC32的设计师们就将MMU换成了更为简单的FMT（Fixed Mapping Translation，固定映射表）模块以及BMX（Bus Matrix，总线矩阵）控制机构。

FMT使PIC32单片机能够遵从其他基于MIPS设计的处理器采用的编程模型，从而能够使用标准化的地址空间。这种固定但又兼容的方案简化了工具和应用程序的设计，也简化了PIC32代码的移植，同时还显著减小了器件的尺寸，进而降低了成本。

BMX给主存储器区的划分带来了灵活度。在CPU提取数据和指令、DMA外围设备以及在线调试器（ICD）逻辑发出访问存储器请求时，BMX还负责总线仲裁。

表6-1列出了相当复杂的转换表以及PIC32MX系列器件的存储器映射。乍一看是有点儿恐怖，但是如果你随我读完下面几段文字，就会发现它很好理解。

表 6-1 PIC32MX 的转换表和存储器映射

存储器类型		虚拟地址		物理地址		字节数
		起始地址	结束地址	起始地址	结束地址	计算
内核地址空间	引导 Flash	0xBFC00000	0XBFC02FFF	0x1FC00000	0x1FC02FFF	12KB
	程序 Flash[1]	0xBD000000	0xBD000000+ BMXPUPBA−1	0x1D000000	0x1D00000+ BMXPUPBA−1	BMXPUPBA
	程序 Flash[2]	0x9D000000	0x9D000000+ BMXPUPBA−1	0x1D000000	0x1D000000+ BMXPUPBA−1	BMXPUPBA
	RAM（数据）	0x80000000	0x80000000+ BMXDKPBA−1	0x00000000	BMXDKPBA−1	BMXDKPBA BMXDKPBA
	RAM（程序）	0x80000000+ BMXDKPBA	0x80000000+ BMXDUDBA−1	BMXDKPBA	BMXDUDBA−1	BMXDUDBA− BMXDKPBA
	外围设备	0xBF800000	0xBF8FFFFF	0x1F800000	0x1F8FFFFF	1MB
用户地址空间	程序 Flash	0x7D000000+ BMXPUPBA	0x7D000000+ PFM Size−1	0xBD000000+ BMXPUBPA	0Xbd000000+ PFM Size−1	PFM Size− BMXPUBPA
	RAM（数据）	0x7F000000+ BMXDUDBA	0x7F000000+ BMXDUPBA−1	0xBF000000+ BMXDUDBA	0xBF000000+ BMXDUPBA−1	BMXDUPBA− BMXDUDBA
	RAM（程序）	0x7F000000+ BMXDUPBA	0x7F000000+ RAM Size[3]−1	0xBF000000+ BMXDUPBA	0xBF000000+ RAM Size[3]−1	DRM Size− BMXDUPBA

1. 不可缓存段（KSEG1）的程序 Flash 虚拟地址。
2. 可缓存和可预取段（KSEG0）的程序 Flash 虚拟地址。
3. RAM 的大小随 PIC32MX 器件的不同而不同。

首先，请找出 PIC32 单片机的主存储器块（RAM 和 Flash 存储器）在 32 位物理寻址空间内的位置（参见图 6-7）。然后检查物理地址列，就会发现 RAM 的起始地址是 0x00000000，而 Flash 存储器则从 0x1D000000 开始。最后，所有外围设备（特殊功能寄存器）都位于从地址 0x1F800000 开始的单元内，还有一部分 12KB 的 Flash 存储器的地址起始于 0x1FC00000，它用作引导区。

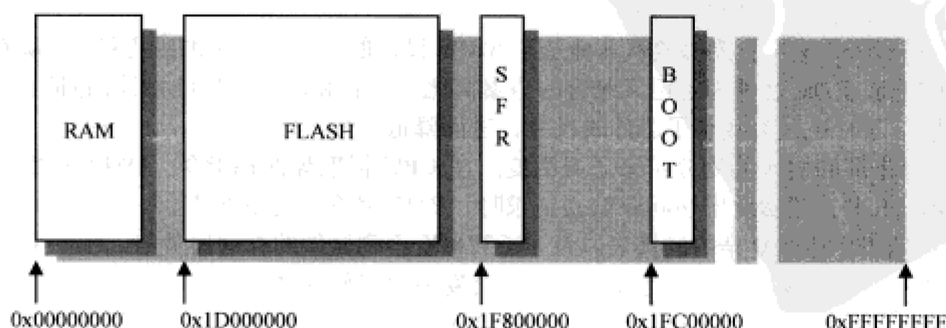


图 6-7 PIC32 单片机的物理寻址空间

根据不同的目的，需要访问上述不同的存储器区域。PIC32 的设计者希望通过隔离不同的存储器确保用户能利用特殊的“规则”来防止普通（编程）错误危害到应用程序。比如，在运行操作系统时，希望能阻止应用程序访问操作系统使用的数据区（RAM）。换句话说，用户代码不允许访问内核数据。BMX 控制模块用于实现第一层操作（如图 6-8 所示）。通过它的一些控制寄存器，能够将主存储区分割成大小可变的多个片断。比如，利用 BMXPUPBA 寄存器可以将部分 Flash 存储器再映射到仅供用户模式使用的物理地址 0xBD000000 甚至更高。类似地，利用寄存器 BMXDKPBA 和 BMXDUDBA 可以将 RAM 数据存储器分割为 4 片，从而将内核数据存储区与用户数据存储区分开，进而还能为那些在 RAM 内运行的程序分割更小的数据存储区。由于 RAM 的访问速度通常比 Flash 存储器快得多，即使考虑使用 cache 也是如此，因此上述方法可以提升系统的性能。

接下来，FMT（或者更常见的情况是 MMU）将给整个系统增添一套复杂的新层，它将所有的物理地址转换成虚拟地址，事情变得有点儿乱。这意味着程序可以运行在两个广阔而独立的地址空间里：一个是用户应用程序使用的、位于 32 位寻址空间低半部分的空间（地址小于 0x80000000），另一个则供基于 MIPS 的处理器标准实现的内核使用（地址大于 0x80000000）。这和表 6-1 中列出的两个部分是相符的，表中前两列显示了在对应工作模式下为每个存储区分配的新的虚拟地址。

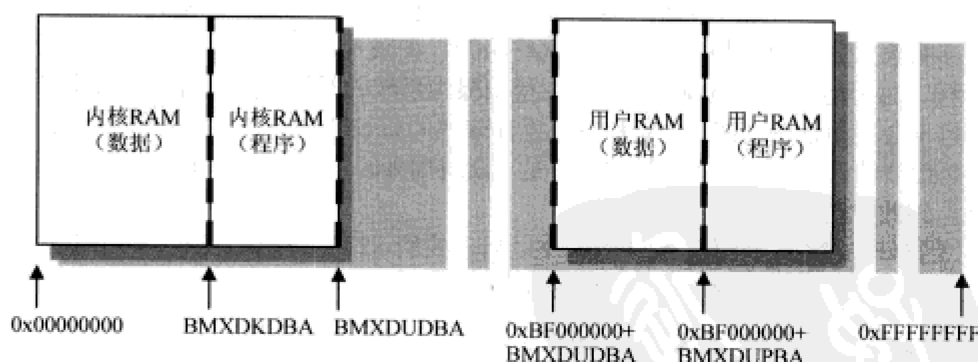


图 6-8 总线矩阵的 RAM 分区

注解 唯一与 MPLAB C32 编译器以及链接器有关的地址就是虚拟地址！这一点在本章前面的内容中就已经提到过。

为了明确起见，图 6-9 给出了一个运行在用户模式下的应用程序的虚拟存储器映射结果。

请注意在用户模式下引导 Flash 存储器根本没有被重映射。没有虚拟地址可供用户程序去访问受保护的区域。无论情况多么糟糕，代码总是运行在用户模式下，它不会危害到基本的操作系统（或者引导器）。

与此类似，外围设备也没映射到用户虚拟地址空间。同样，无论用户代码出现多么严重的问题，它都不会触及硬件，修改或者破坏器件配置。

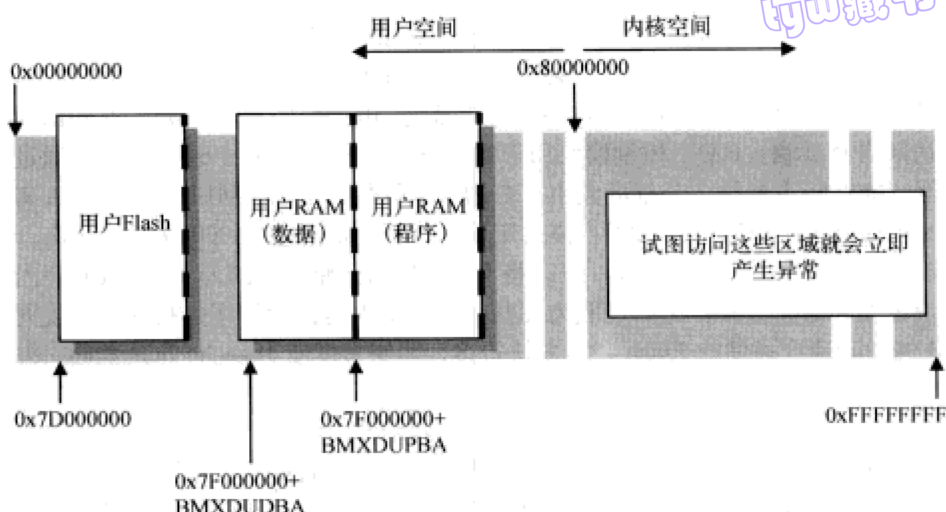


图 6-9 用户模型下的虚拟存储器映射

6.10 嵌入式控制应用的存储器映射

如果你打算使用一个带有大量功能的重量级的操作系统，那么尽管看起来很好、很花哨，但是在大多数嵌入式控制应用中并不需要这些功能。在嵌入式应用中，所有的代码很可能和操作系统一样，总是运行在内核模式。甚至在使用操作系统的时候，会发现大多数实时操作系统（RTOS）也不会使用这些功能，它对执行速度和效率的喜好胜过"保护"。对于嵌入式控制系统来说，这是理智的选择。嵌入式应用程序都是"众所周知的"，应该具备较好的鲁棒性并经过良好的测试，因此也是信得过的。

这是个很好的消息，因为这意味着从今以后，我们可以完全忽略表 6-1 的下半部分，将精力集中在内核模式的虚拟映射上（参见图 6-10）！

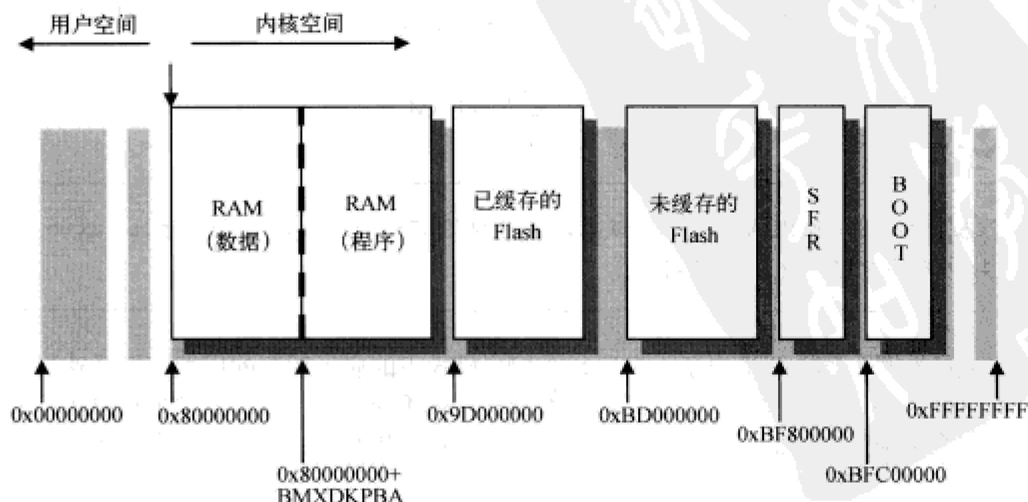


图 6-10 用户模型下的虚拟存储器映射

最后一个需要说明的是内核 Flash 程序存储器使用两套虚拟地址空间的原因。在 MIPS 文

献中它们传统上被记为 *kseg0* 和 *kseg1*。如果看一下表 6-1 中的物理地址列，就会注意到它们最终都指向相同的物理存储器空间。它们的区别只是存储器 cache 管理虚拟地址的方法不同。如果程序在第一套虚拟地址空间（*kseg1*）中运行，那么存储器级 cache 自动关闭。相反，位于 *kseg0* 中的部分代码则将由 cache 访问。在后面几章，我们将了解更多有关这样做的原因及其对代码性能的影响。

6.11 小结

在本章中，我们首先快速回顾了基本的字符串声明与操作，然后简要了解了使用指针和动态存储区分配的方法，还学习了通过.map 文件掌握应用程序是在哪里以及如何使用 PIC32 的存储器的。此外，我们还研究了 PIC32 的总线矩阵模块，并且学习了它是如何极为灵活地控制 Flash 和 RAM 存储器的分割与访问的。尽管很多嵌入式控制应用系统都只使用最基本的（默认的）配置，但是 PIC32MX 架构提供的标准地址空间布局也能使它与很多已有的 MIPS 架构的工具和操作系统兼容。

6.12 对 C 语言行家的提示

在 C 语言中，字符串被看作简单的字符数组。C 语言也没有不同存储区的概念（RAM 或 Flash）。C 语言中的 `const` 属性经常与大部分其他变量类型一同使用，它只是指示编译器捕捉普通的参数使用错误。当利用函数传递带有 `const` 属性的参数或者某个变量被声明为 `const` 时，编译器事实上只会在后面试图修改它们时给出提示标志。而 MPLAB C32 编译器则自然地扩展了这个功能，它能够提醒编译器和链接器更有效地利用存储器资源。

6.13 对汇编语言行家的提示

string.h 函数库包含很多有用的块操作函数，通过使用指针可以使它们操作任何类型的数据数组，而不仅仅是字符串。这些函数是：

- `memcpy()`，将存储器块的内容复制到新地址；
- `memmove()`，将某个存储器块的内容移动到新位置；
- `memcmp()`，比较两个存储器块的内容；
- `memset()`，初始化某个存储器块。

相反，ctype.h 函数库包含的函数则可以根据字符在 ASCII 表中的位置来区分字符，区分大小写，或者进行大小写转换。

6.14 对 PIC 单片机行家的提示

由于 PIC32MX 的程序存储器采用（单电压）Flash 工艺实现，允许在运行代码时进行编程，因此我们可以设计出基于引导器的应用程序，即应用程序可以自动“更新”自己的部分或全部代码。

我们还可以将部分 Flash 程序存储器用作 NVM（NonVolatile Memory，非易失性存储器）。尽管有一些糟糕的基本限制，比如，Flash 存储只能先进行大块删除，每个大块称为页（page），它包括 1024 个字。之后，才能一次写入一个字或者写入一个称为行（row）的小块，每行又包括 128 个字。

PIC32 外围设备函数库对我们很有帮助，它提供了一组专门用于操作片上 Flash 存储器的函数（NVM.h）。其中，`NVMProgram()` 可能是功能最强大的函数，它能从给定的虚拟地址起

写入任意长度的数据块，并能在穿越页边界时自动完成必要的分割。

6.15 提示与技巧

一旦学会了如何用零终止符提高字符处理的效率，用 C 语言进行字符串操作就变得有趣了。以 mycpy() 函数为例：

```
void mycpy( char *dest, char * src)
{
  while( *dest++ = *src++);
}
```

这是一段极其危险的代码，因为它不限制可以复制多少字符，并且不检查指针 dest 指向的缓冲区是否足够大。于是你可以想象，一旦字符串 src 丢失终止符，将会发生什么情况。这段代码很容易超出已分配的变量空间，并能破坏整个数据存储器的内容。指针真强大啊！

不久我们将研究 DMA 模块并会发现它能共享 PIC32 的存储器总线，从而能在存储器与外围设备之间实现快速数据传输。我们还将研究使用 DMA 模块在不同的存储器缓冲区之间高效地移动大块数据。事实上，PIC32 的外围设备函数库中有一些 DMA 函数专门用于 DMA 通道的字符串和块操作，包括 DmaChnMemcpy()、DmaChnStrcpy()以及 DmaChnStrncpy()。在该函数库中还可以找到函数 DmaChnMemCrc()，它并不用于传输数据，而是为给定的一段数据（不管它有多长）添加 CRC 码。或者，DMA 模块在进行数据传输时通过调用 CrcAttachChannel()函数也能自动完成 CRC 计算。

6.16 练习

请开发一个新的字符串操作函数，完成以下操作：

(1) 在一个字符串数组中顺序搜索一个字符串；

(2) 实现二进制字符串搜索；

(3) 开发一个简单的散列表（hash table）管理函数库。

6.17 参考书

N. Wirth 所著的 *Algorithms+Data Structures=Programs*。Pascal 语言之父 Wirth 将以非常简洁易懂的方式带领你从基本的编程开始直到编写自己的编译器。有人告诉我，这本书现在不容易找到了。但是，无论找到这本书有多难，我保证你一定会觉得找到它是值得的！

6.18 链接

http://en.wikipedia.org/wiki/Pointers#Support_in_various_programming_languages。在这里你可以学到更多有关指针的知识，并能看到它们是如何用于各种编程语言的。

第二部分

实　　践

祝贺你！你已经坚持完成了前6章的探索并获得了信心，能够用MPLAB PIC32软件工具包创建简单的工程了。在接下来的第二部分课程中，还有更多惊喜等待着你！

在本书的第二部分，我们将继续逐个研究那些能使PIC32单片机与外界其他设备相连接的基本外围设备。由于这些实例的难度稍大，因此强烈建议你准备一块PIC32芯片，以便能够测试这些实际的工程实例。事实上，只要有一块带有PIM适配器或实际的PIC32MX处理器模块（PIM）的PIC32 Starter Kit以及任何一款兼容的在线调试器就可以了。尽管书中会经常提到Explorer 16演示板，但其实任何具备类似功能或者能提供小型原型板区的第三方工具都能有效地完成本部分的研究任务。

第 7 章　时间与初始化

7.1　计划

在前 6 章中，通过对嵌入式控制系统，特别是对 PIC32MX 架构进行 C 语言编程，逐步回顾了 C 语言编程的大部分基本概念。我们还初步熟悉了影响 PIC32 性能的基本部件，比如 32 位乘法器、中断系统、寄存器集以及存储器管理模块。但是到目前为止，我们还只是在反汇编窗口中计算汇编指令的数目，或者利用 MPLAB SIM 仿真器的 StopWatch 窗口计算指令周期数。在分析代码的执行过程时，我们始终避免直接提到代码的执行时间，必要时则使用外围设备（定时器）来实现延时。即使是在讨论中断或者比较各种数据类型的效率时，也没有仔细讨论它们与代码实际执行速度的复杂关系。这么做的目的是将不同的问题相互分开，并控制内容的复杂度，使其逐渐增大。然而，在了解 PIC32“跑”得到底有多快之前，还必须研究两个新的关键系统：时钟系统和存储器缓存系统。它们都是 PIC 架构新增的，同时也是为了将 PIC32 引擎微调到最佳性能所必须掌握的。

7.2　准备

本章中除了要使用基本的软件工具 MPLAB IDE 以及 MPLAB C32 编译器之外，还需要实际的硬件，以便进行实验。无论是 PIC32 入门开发板，还是与 Explorer 16 演示板相连的其他在线调试器都可以。当然，还需要能够在所选硬件平台上运行的实际芯片 PIC32MX。

利用 New Project Setup（创建新工程）检查表创建名为 Running 的新工程，并创建名为 running.c 的源文件。

7.3　探索

首先看一下 PIC32MX 系列的主时钟电路。如图 7-1 所示，这部分硬件有些复杂，因此需要花些时间来熟悉它。

对于熟悉以前的 8 位 PIC 单片机的读者来说，图 7-1 的大部分模块看起来都有些眼熟。而熟悉 dsPIC33 特别是 PIC24H 系列单片机的读者则会觉得图 7-1 特别熟悉！这当然不是巧合。从第一代的 PIC16C54 起，所有的 PIC 单片机都包含灵活的振荡器电路，这种灵活性已经一代接一代地继承下来，并逐步演化成现在 PIC32MX 的形式。下面来看看我们都可以做些什么，更重要的是要理解为什么可以这么做！

图 7-1 中框图左侧有 5 个振荡器/时钟源。其中有 2 个采用内部振荡器，其他 3 个则需要外部晶振或者振荡器电路。

- ❑ 内部振荡器（FRC）。用于低功耗高速运行模式，无需外部元件，在校正后可以提供较为准确的 8MHz 时钟（±2%）。
- ❑ 内部低频低功耗振荡器（LPRC）。用于低功耗低速运行模式，需要外部元件，提供基本（低精度）的 32kHz 时钟。
- ❑ 外部主振荡器（POSC）。用于高速高精度（基于石英晶体）运行模式，可以直接与高达 20MHz 的晶振相接（与 OSCI、OSCO 引脚相接），并有两种增益设置可选：适用于低于 10MHz 的石英晶体的 XT 模式，适用于高于 10MHz 的石英晶体的 HS 模式。

- ❑ 外部低频、低功耗振荡器（也称为辅助振荡器，SOSC）。用于低速低功耗运行模式，与外部的 32 768Hz 晶体相接，可用作整个芯片的主时钟或者仅用作定时器 1 和 RTCC 模块的时钟源。它的精度很高，因此是需要精确时间保持（timekeeping）应用的理想时钟源。
- ❑ 外部时钟源（EC）。该模式使用外部电路代替振荡器，能为单片机提供任意频率的方波输入信号。

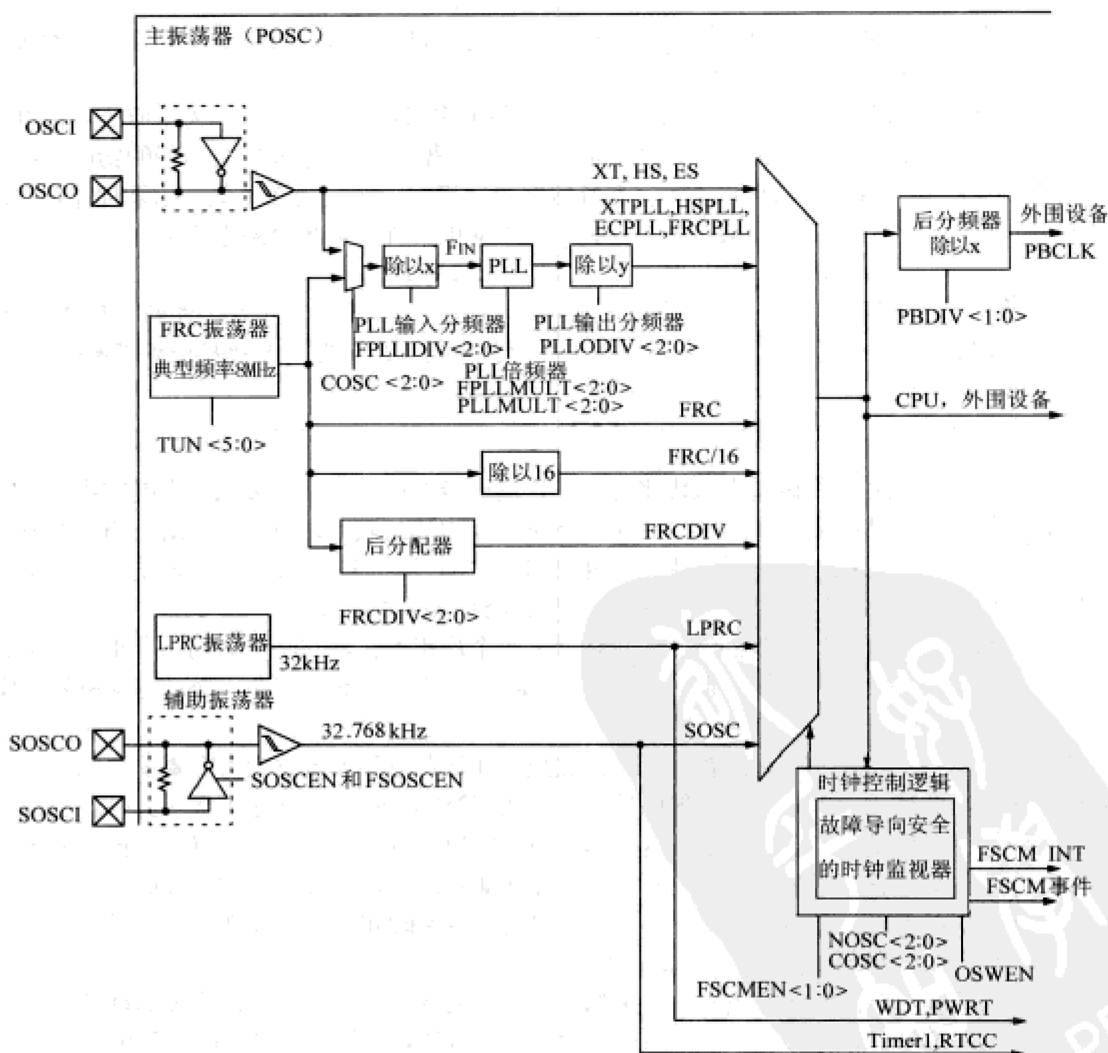


图 7-1 PIC32MX 的时钟框图

这 5 种时钟源可作为基本选择，能为单片机提供所需频率、功耗和精度的输入时钟信号，框图右侧的后续电路还可以实现更多处理。事实上，每个时钟源提供的时钟还可以进一步乘以或除以一定的数值，实现更宽的频率范围。

7.4 性能与功耗

本书不会去介绍每种时钟源的所有功能，但是你必须理解 PIC32 的设计师为何花费巨大的精力使芯片具备如此众多的途径来产生简单的方波。

在嵌入式控制以及消费类应用系统中，无论应用系统是便携式的（电池供电的）还是由某种专用电源供电的，都必须满足如下两个重要的约束。

- 功耗决定了需要设计的电源电路的尺寸和成本。如果采用电池供电，系统的功耗就决定了电池的大小和成本，或者反之，电池的大小决定了应用系统的寿命（运行时间）。
- 所测得的性能决定应用系统在给定时间内能完成的工作量。对于实时应用，该参数可能就是瓶颈。

通常情况下，在嵌入式控制应用设计中，上述两个约束是相互矛盾的。为了使给定的电路完成更多的工作，我们希望采用最高的时钟频率。但是根据 CMOS 集成电路运行的物理定律可知，时钟频率越高，器件的功耗也越大。并且这两个因素实际上成线性关系：如果时钟频率提高一倍，那么芯片的处理能力就会提高一倍，但是对应的器件功耗也会随之增大。



注解 器件的时钟频率翻倍，但是功耗并不翻倍。原因是，CMOS 器件的功耗包括静态功耗和动态功耗。其中，静态功耗与时钟频率无关，只有动态功耗会随频率的提高而成倍增大。

PIC32 内部可以并已经做过大量优化处理，这使得器件在任何功耗级别下都能完成最大的工作量。比如，PIC32MX 的数据手册（在编写本书时获得的最新的数据手册）给出的器件电气特性显示，当器件运行在 4MHz 时，典型的电流消耗为 11mA（3.3V，25℃）。但是当器件工作于 72MHz 并保持其他条件不变时，器件的电流消耗将增加为 64mA。

尽管参数就是如此，但是我们仍然有责任保证每个应用的性能与功耗达到一个平衡，从而使成本最小化，减小尺寸，或者只是使电池的寿命最大。（此外，再让我加一条“与温室效应作战”吧！）

事实上，有些任务在 4MHz 时钟下就可以完成，并且考虑到大多数应用需要单片机在不同时间运行在不同的模式，因此让应用程序始终运行在 72MHz 时钟下就毫无意义。尽管这看起来可能有些过分，但是它可以在像手机这样的应用中实现并行处理。我们知道，手机在大多数时间里都处于待机状态，只是等待按键唤醒它。在其他时间里，它可能只需完成简单的功能，比如搜索联系簿或者更新内存的信息。因此，它只需在少量时间内进行复杂的数字运算、数字信号处理并执行压缩和解压音频输入和输出流的算法。

很多嵌入式控制（以及消费类）应用中都有类似的要求，并且时钟电路的灵活性越高，对应用程序的功耗控制得就越好。为了帮助用户拥有最完备的功耗管理手段，PIC32 单片机的时钟模块具备以下特性：

- 实时切换内部和外部振荡源；
- 实时控制时钟分配器；
- 实时控制 PLL 电路（时钟乘法器）；
- 空闲（IDLE）模式，此时 CPU 停止，只有个别外围设备继续运行；
- 休眠（SLEEP）模式，此时 CPU 和外围设备都停止，并等待特殊的（一组）事件唤醒；

- 独立控制的外围设备时钟（PBCLK），从而使得当 CPU 需要运行在高频时钟下时，外围设备模块的功耗仍能被优化得较低。

7.5 主振荡时钟链

之所以要首先介绍主振荡时钟链，是因为它最为普通，并且在下面几章中，我们还将开发一些需要高性能或者高精度时钟的演示工程。正如你所看到的，在 Explorer 16 演示板和 PIC32 Starter Kit 中，OSCI 和 OSCO 管脚上都接有一个 8MHz 的晶振。在该频率下（低于 10MHz），建议将主振荡器设置为工作在 XT 模式。

根据应用的不同，我们将立刻面临两种选择。一种是直接使用 8MHz 信号，一种是将 8MHz 信号作为倍频器（PLL）电路的输入。很明显这里要选择后者。然而在介绍这些内容之前，我们有必要再学习一些有关 PLL 电路的知识。

尽管 PLL（Phase Locked Loop，锁相环）电路比较复杂，但是 PIC32 的设计师已经设法向用户隐藏了 PLL 电路的复杂性，并且只要求用户遵守一些简单的规则。首先，需要向它提供一定输入频率范围（小于 4MHz）的时钟信号。其次，在执行代码以及实现时钟同步前，需要给它一定的稳定或者“锁定”时间。此外，通过简单配置（通过图 7-2 所示的 OSCCON 寄存器）就能选择倍频系数（PLLMULT），并验证是否已正确锁定（SLOCK）。

U-0	U-0	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x
—	—	PLLODIV <2:0>			FRCDIV <2:0>		
位31							位24

R/W-0	R-0	U-0	R/W-x	R/W-x	R/W-x	R/W-x	R/W-x
DRMEN	SOSCRDY	—	PBDIV <1:0>		PLLMULT <2:0>		
位23							位16

U-0	R-0	R-0	R-0	U-0	R/W-x	R/W-x	R/W-x
—	COSC <2:0>			—	NOSC <2:0>		
位15							位8

R/W-0	r-0	R-0	R/W-0	R/W-0	r-0	R/W-0	R/W-0
CLKLOCK	—	SLOCK	SLPEN	CF	—	SOSCEN	OSWEN
位7							位0

图 7-2 OSCCON 寄存器

因此，当使用 Explorer 16 板或者 PIC32 Starter Kit 时，为了遵守第一条规则，就需要将输入时钟频率由 8MHz 降低为 4MHz。从图 7-1 中的框图或者图 7-3 中的简化图可以看出输入分频器是如何轻松地实现第一次降频的。

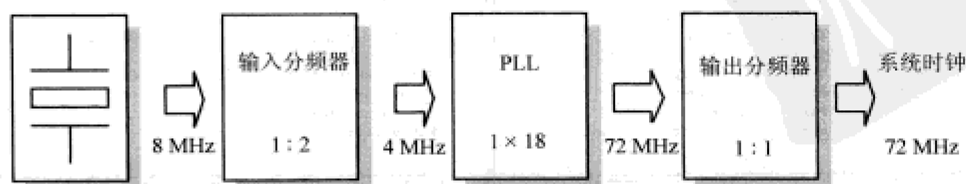


图 7-3 主振荡器时钟链

PLL 的倍频系数可以通过 PLLMULT 位控制，可选的范围是 15 倍至 24 倍。由于 PIC32MX 的最高运行频率被限制为 75MHz（截至编写本书时），因此选择倍频因子 18×就可以得到最接近器件运行条件的频率 72MHz。此外，输出分频器模块还为我们提供了控制时钟频率的最后机会。当需要最佳性能时，就可以将输出分频器设为 1:1。如果应用需要，还能通过输出分频来降低功耗，输出分频最低可达 1:256，此时对应的输出频率约为 280kHz。如果需要更低的频率，那么使用辅助振荡器（SOSC）会更好，它的工作范围是 32~100kHz；或者使用低功耗内部振荡器（LPRC），它的工作频率约为 32kHz。作为参考，最新的数据手册显示，PIC32 使用 LPRC 时的典型功耗仅为 200μA！

7.6　外围设备总线时钟

作为另一种优化应用系统的性能和功耗的方法，PIC32 为所有外围设备都提供了独立的时钟信号。通过将系统时钟发送至另一个分频电路（由图 7-3 所示的模块链进一步扩展），就能产生外围设备总线（PB）时钟信号。高速处理器通常需要在定时器前使用大分频系数以获得所需的定时时间，另外，串行端口也需要使用大波特率分频器（通常遇到的是这种情况）。幸亏有外围设备总线分频器，当处理器自由地运行在最高速度时，还可以减小外围设备总线消耗的能量。

该特性由 PBDIV 位控制，它也位于 OSCCON 寄存器内。我们一直以来使用的并将在后续工程中继续使用的外围设备总线频率为 36MHz，对应的外围设备总线时钟与系统时钟之比为 1:2。

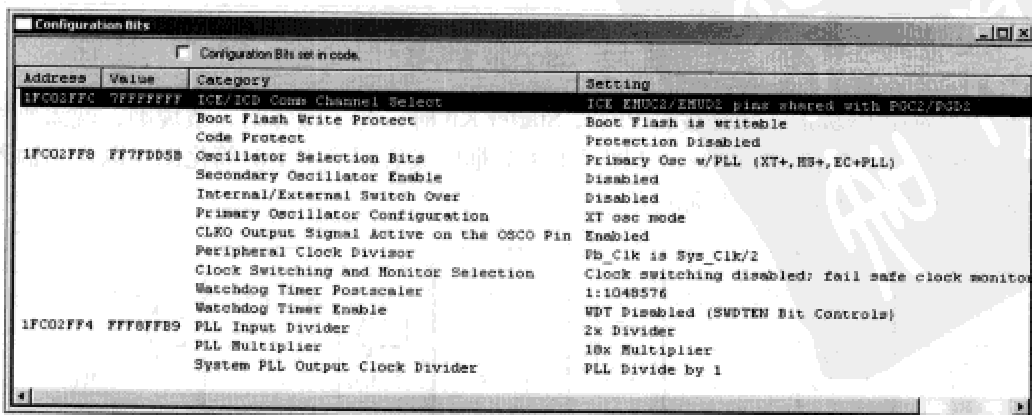
7.7　器件的初始配置

允许在器件运行时控制时钟为我们提供了强大的功耗控制能力，但是当器件刚刚上电并被初次激活时又会发生什么呢？

你可能已经知道，在 PIC32 的非易失性存储器（Flash）中有一组被称为配置位的数据位，它们被用作器件的初始配置。振荡器模块则利用其中的一些位作为 OSCCON 寄存器的初始配置。我们可以通过 MPLAB 的 Configure|Configuration Bits…菜单设置这些配置位。

下面回顾一下从开始使用 Device Configuration（器件配置）检查表以来就推荐你完成的配置。

图 7-4 就是我为本书所有练习推荐的配置。它包括下列选项（依照振荡器配置的重要性排序）。



Configuration Bits

Configuration Bits set in code.

Address	Value	Category	Setting
1FC02FFC	7FFFFFFF	ICE/ICD Comm Channel Select	ICE EMUC2/EMUD2 pins shared with PGC2/PGD2
		Boot Flash Write Protect	Boot Flash is writable
		Code Protect	Protection Disabled
1FC02FF8	FF7FDD5B	Oscillator Selection Bits	Primary Osc w/PLL (XT+,HS+,EC+PLL)
		Secondary Oscillator Enable	Disabled
		Internal/External Switch Over	Disabled
		Primary Oscillator Configuration	XT osc mode
		CLKO Output Signal Active on the OSCO Pin	Enabled
		Peripheral Clock Divisor	Pb_Clk is Sys_Clk/2
		Clock Switching and Monitor Selection	Clock switching disabled; fail safe clock monito
		Watchdog Timer Postscaler	1:1048576
		Watchdog Timer Enable	WDT Disabled (SWDTEN Bit Controls)
1FC02FF4	FFF8FFB9	PLL Input Divider	2x Divider
		PLL Multiplier	18x Multiplier
		System PLL Output Clock Divider	PLL Divide by 1

图 7-4　Device Configuration 对话框

(1) 使用带有 PLL 电路的主振荡器。
(2) 设置主振荡器工作在 XT 模式。
(3) 设置 PLL 输入分频系数为 1:2（以产生 4MHz 输入）。
(4) 设置 PLL 倍频系数为 18×。
(5) 设置 PLL 输出分频系数为 1:1（以产生 72MHz 系统时钟输出）。
(6) 设置外围设备时钟分频系数为 1:2（以产生 36MHz 外围设备总线时钟输出）。

为完成配置，还需要下面的附加选项。

(7) 打开时钟输出；当使用任何内部振荡器来控制额外的 I/O 管脚时，可以关闭该选项。
(8) 关闭辅助振荡器（还可以在稍后器件运行时再打开它）。
(9) 关闭内部/外部振荡器切换功能。（尽管本书的所有练习中都只使用外部晶体，但是你也可以尝试其他设置。）

最后，在调试和开发时还需使用下列选项。

(10) 如果使用 ICD/ICSP 接口，请共享 DBG2 和 PGM2。（这取决于你选择的在线调试器。）
(11) 允许修改引导 Flash（Bootloader 写保护关闭）。
(12) 关闭代码保护功能（至少在开发阶段要关闭）。
(13) 关闭看门狗定时器。
(14) 关闭时钟切换和故障导向安全的时钟监视器。

一旦完成设置，这些配置位就被存储在工作空间文件（.mcw）中，并将在每次利用编程器向器件烧录新代码时被烧录到器件配置位中。

比较图 7-2 和图 7-4 就会发现，PLL 输入分频器的值将以配置位选项的形式出现，但是无法通过 OSCCON 寄存器进行修改。只要仔细考虑一下就会发现，这样的设计是合理的。由于外部晶振无法改变（除非从 PCB 上拆除旧晶振并安装一个不同频率的新晶振），因此没有必要在运行时改变输入分频器的值。如果配置位中设定的值不正确，那么 PLL 倍频器将无法工作，PIC32 也无法执行代码。

7.8 在代码中设定配置位

为了使工程代码自成归档，并且避免未来出现任何差错（比如丢失工程文件或者应用系统的源文件使用了错误的配置），MPLAB C32 编译器还提供设定器件配置位的附加功能。它通过`#pragma config`命令实现。

由于不同器件的配置位的数量和取值不同，因此 MPLAB 为每款 PIC32 单片机提供了一系列选项，作为帮助系统的一部分。选择 Help|Topic 打开帮助系统选择对话框，单击 PIC32MX Config Setting（PIC32MX 配置设定），具体请参见图 7-5。

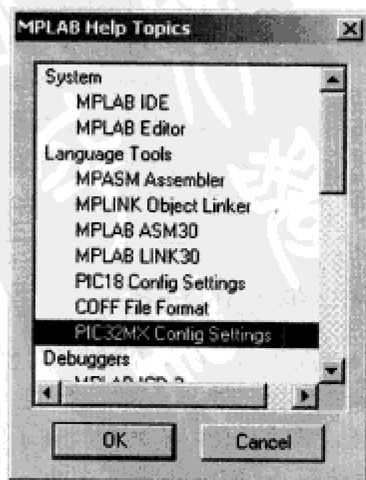


图 7-5 MPLAB Help Topics 对话框

选择你所使用的器件型号 PIC32MX360F512L，然后确定每个配置位使用的正确语法。表 7-1 给出了 PLL 输出分频器的例子。

表 7-1　PLL 输出分频器的取值

FPLLODIV=DIV_1	除以 1
FPLLODIV=DIV_2	除以 2
FPLLODIV=DIV_4	除以 4
FPLLODIV=DIV_8	除以 8
FPLLODIV=DIV_16	除以 16
FPLLODIV=DIV_32	除以 32
FPLLODIV=DIV_64	除以 64
FPLLODIV=DIV_256	除以 256

一条#pragma config 语句里可以通过逗号分割设置多个配置位，比如在下面的例子中，我们将再次配置前面提过的振荡器设置：

```
#pragma config POSCMOD=XT, FNOSC=PRIPLL
#pragma config FPLLIDIV=DIV_2, FPLLMUL=MUL_18, FPLLODIV=DIV_1
```

请注意，对于#pragma 语句未配置的参数，则使用器件数据手册中的默认值。

下面将再写一条#pragma 语句，完成对外围设备总线时钟分频器的配置，关闭看门狗和代码保护，并允许对引导存储器编程。后面的工程都需要这样配置（至少在开发阶段需要这样配置）：

```
#pragma config FPBDIV=DIV_2, FWDTEN=OFF, CP=OFF, BWP=OFF
```

建议你将这段代码放在包含工程主函数的源文件的最顶端。

为了避免与 MPLAB Configuration Bits 对话框（参见图 7-4）中的设置冲突，请务必选中 Configuration Bits Set in Code 选项框。



注解　当选中 Configuration Bits Set in Code 选项框时，对话框内的所有内容将变为灰色。这是每个新工程的默认选择。请当心，如果忘记在代码中设置#pragma config 语句，那么将使用器件数据手册中说明的默认配置。该默认配置可保证器件“安全”运行，但是其中大多数设定在开发阶段使用会发生冲突或者导致错误。本书前面几章中未在代码中设置配置位，目的是为了避免你对代码“分心”，并且避免一开始就声明很多内容。从现在开始，你可以根据自己的喜好来选择了！

7.9　艰巨的任务

下面要写一些复杂的程序，然后将程序烧录在 PIC32 Starter Kit 或者 Explorer 16 演示板中，以便测试 PIC32MX 的实际性能。

看看我在代码文档中找到了什么！在我硬盘的偏僻子目录中，找到了能追忆在大学里学习数字信号处理的旧时光的东西，于是我编写了下面的程序：

```
// input vector
unsigned char inB[N_FFT];

// input complex vector
float  xr[N_FFT];
```

```
float  xi[N_FFT];

// Fast Fourier Transformation
void FFT(void)
{
  int    m, k, i, j;
  float  a, b, c, d, wwr, wwi, pr, pi;

    // FFT loop
    m = N_FFT/2;
    j = 0;
    while(m > 0)
    { /* log(N) cycle */
      k = 0;
      while(k < N_FFT)
      { // batterflies loop
     for(i = 0; i < m; i++)
     { // batterfly
       a = xr[i+k];       b = xi[i+k];
       c = xr[i+k+m];       d = xi[i+k+m];
       wwr=wr[i<<j];  wwi = wi[i<<j];
       pr=a-c;  pi = b-d;

         xr[i+k]     = a + c;
         xi[i+k]     = b + d;
         xr[i+k+m]   = pr * wwr - pi * wwi;
         xi[i+k+m]   = pr * wwi + pi * wwr;
       } // for i
     k += m<<1;
     } // while k
     m >>= 1;
     j++;
   } // while m
} // FFT
```

这是一个快速傅里叶变换（FFT）函数，它是数字信号处理中最常用的工具，而这里只是它的简化形式，并用于处理一组长度为2的幂次方的样值。该FFT算法能高效地运算离散傅里叶变换（DFT）及其反变换，也就是说，它可以将信号由时域表示转换为频域表示。换句话说，如果将表示输入信号样本的一个数组数据（inB[]）作为FFT函数的输入，该函数将返回一个新的数组，其数值对应输入信号每个谐波（正弦分量）的幅值，即信号的频谱（frequency spectrum）。FFT在很多应用中都有重要作用，不但在数字信号处理中，而且在求解偏微分方程以及大整数快速乘法的算法中都有应用。关于如何优化FFT以及确定对给定的数据集合进行处理所需的最小运算量等方面的研究很多。但是，我们并不打算在这里研究如何优化算法，而是使用“学究式”的实现作为高强度浮点运算算法的实例，以完成性能测试。

实际上，前面给出的算法只是完整的离散傅里叶变换实现所需的一部分工作。为了获得必要的精度，输入数据集必须在使用前先被加窗（windowed）。可以把它看作是突然切断一段输入信号，并且需要添补数据以圆滑算法的响应中信号末端的陡峭边沿：

```
// apply Hann window to input vector
void windowFFT(unsigned char *s)
{
  int i;
  float *xrp, *xip, *wwp;
```

```
  // apply window to input signal
  xrp = xr; xip = xi; wwp = ww;
  for(i=0; i<N_FFT; i++)
  {
    *xrp++ = (*s++ - 128) * (*wwp++);
    *xip++ = 0;
  } // for i
} // windowFFT
```

在完成FFT处理后，必须取走（复数）输出单元并进行适当的缩放，然后更新输入数组：

```
void powerScale(unsigned char *r)
{
  int i, j;
  float t, max;
  float xrp, xip;

  // compute signal power (in place) and find maximum
  max = 0;
  for(i=0; i<N_FFT/2; i++)
  {
    j = rev[i];
    xrp = xr[j];
    xip = xi[j];
    t = xrp*xrp + xip*xip;
    xr[j] = t;
    if (t>max)
      max = t;
  }

  // bit reversal, scaling of output vector as unsigned char
  max = 255.0/max;
  for(i=0; i<N_FFT/2; i++)
  {
    t = xr[rev[i]] * max;
    *r++ = t;
  }
} // powerScale
```

为了实现流水线处理并且避免执行效率明显降低，通常会提前完成最少的内务处理，即只初始化一些被反复使用的数组，比如所谓的*旋转数组*（rotation array）、*加窗数组*（window array）以及*位反数组*（bit reversal array）。下面是它们的定义方法以及可供使用的初始化函数：

```
// input vector
unsigned char inB[N_FFT];
volatile int inCount;

// rotation vectors
float  wr[N_FFT/2];
float  wi[N_FFT/2];

// bit reversal vector
short  rev[N_FFT/2];

// window
float  ww[N_FFT];
void initFFT(void)
{
```

```
    int i, m, t, k;
    float *wwp;

    for(i=0; i<N_FFT/2; i++)
    {
      // rotations
      wr[i] = cos(PI2N * i);
      wi[i] = sin(PI2N * i);

      // bit reversal
      t = i;
      m = 0;
      k = N_FFT-1;
      while (k>0)
     {
       m = (m << 1)+(t & 1);
       t = t >> 1;
       k = k >> 1;
     }
     rev[i]=m;
   } // for i

   // initialize Hanning window vector
   for(wwp=ww, i=0; i<N_FFT; i++)
     *wwp++ = 0.5 - 0.5 * cos(PI2N * i);

} // initFFT
```

你被吓到了？晕了？千万别这样。别管这些代码，它们实在太复杂了。输入数组的样值个数 N_FFT 越大，PIC32 的运算量也越大。

下面，我们需要做的是将上述代码封装成名为 fft.c 的源文件，并将该文件添加进新工程 Running。

为了使程序看起来更简洁，可以创建一个名为 fft.h 的头文件，并在其中定义 fft.c 文件需要使用的所有符号。

```
/*
**  FFT.h
**
**  power of two optimized algorithm
*/

#include <math.h>

#define N_FFT  256                  // samples must be power of 2
#define PI2N   2 * M_PI/N_FFT

extern unsigned char inB[];
extern volatile int inCount;

// preparation of the rotation vectors
void initFFT(void);

// input window
void windowFFT(unsigned char *source);

// fast fourier transform
void FFT(void);
// compute power and scale output
void powerScale(unsigned char *dest);
```

请将 fft.h 文件添加到 Running 工程的头文件目录中。

下面我们要创建工程的主源文件。将它命名为 run.c 怎么样？（参见图 7-6。）

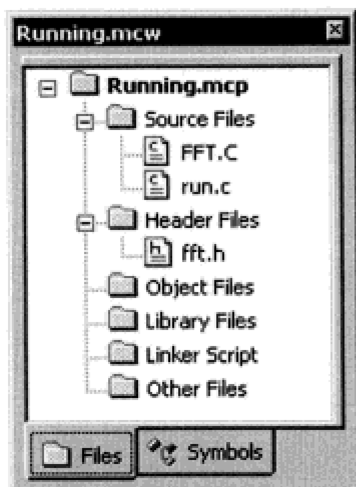


图 7-6 Running 工程的 Project 窗口

为了清晰起见，请在源文件的最顶部添加设定配置位代码。然后引用 fft.h 文件，以便稍后能够调用其中的函数：

```
/*
** Run.c
**
*/
// configuration bit settings
#pragma config POSCMOD=XT, FNOSC=PRIPLL
#pragma config FPLLIDIV=DIV_2, FPLLMUL=MUL_18, FPLLODIV=DIV_1
#pragma config FWDTEN=OFF, CP=OFF, BWP=OFF
#include <p32xxxx.h>
#include <plib.h>
#include "fft.h"
```

下面将创建主函数，依次完成如下功能。

(1) 初始化。

① 首先调用函数 initFFT()。

② 向输入缓冲器（inB[]）填写数据作为测试信号，为简单起见取正弦信号：

```
main()
{
    int i, t;
    double f;

  // 1. initializations
     initFFT();

  // test sinusoid
     for (i=0; i<N_FFT; i++)
     {
        f = sin(2 * PI2N * i);
        inB[i] = 128+(unsigned char) (120.0 * f);
  } // for
```

(2) 执行实际的 FFT 算法，依次调用下面 3 个函数：

```
// 2. perform FFT algorithm
  windowFFT(inB);
  FFT();
  powerScale(inB);
```

(3) 完成处理后就可以进入主（无限）循环。

```
  // 3. infinite loop
  while( 1);
} // main
```

7.10 准备、设置、出发

现在我们就可以生成工程、烧录器件，然后利用断点和手动的 StopWatch 功能获取所需的实际处理时间。然而，这个过程可能极其枯燥并且测量结果并不准确。我有个更好的主意：为什么不使用 PIC32 自带的定时器呢？

为此，我们将再次使用一个 16 位定时器，并将首次利用“一对”定时器模块组成 32 位定时器进行实验。既可以使用定时器 2 和定时器 3 组成一对，也可以用定时器 4 和定时器 5 组成一对。下面的示例就使用后一种组合对 FFT 程序的执行时间计时。

```
// init 32-bit timer4/5
    OpenTimer45(T4_ON | T4_SOURCE_INT, 0);
  // clear the 32-bit timer count
    WriteTimer45(0);

// insert FFT function calls here

  // read the timer count
  t = ReadTimer45();
```

请注意，这里使用了 timer.h 函数库里的函数，因此要在这段程序的最顶端引用 plib.h 文件，它会立即自动引用所有的外围设备函数库。

函数 OpenTimerXX()用于配置定时器，选择时钟源和预分频系数。它等效于在前面的例子中直接向 TxCON 寄存器赋值，但是可读性更好。其主要缺点，也是外围设备函数库通常的缺陷，就是无法在器件的数据手册中描述定时器模块的地方直接找到可用的参数（比如 T4_SOURCE_INT)，因而不得不依赖于单独的文件（函数库手册），有时还得独自查阅头文件，比如这里的 timer.h。事实上，通过查阅头文件（可以用 MPLAB 的编辑器打开它们）就能学会当成对使用定时器时，如何利用初始化函数向定时器对（本例中是 T4）中的第一个模块传递正确的参数。

正如你所预想的，函数 WriteTimerXX()用于设置初始计数值，并有效启动 StopWatch 计时器，而函数 ReadTimerXX()则用于读取 32 位定时器的计数值。它不会停止 StopWatch 计时器，但是会在发出命令的那个精确时刻读取数值，这正是我们需要的。

下面通过选择 View|Watch 菜单打开 Watch 窗口，并在其中添加符号 t。除非已经将查看窗口配置成采用十进制作为默认的显示格式，否则就必须在符号 t 上方单击鼠标，在弹出的上下文菜单中选择 Properties（属性）。请选择 Decimal（十进制）作为变量的默认表示格式。

现在将准备生成工程，并利用开发工具对器件编程。在包含无限循环的那一行设置断点，并单击 Run 按钮，然后就可以坐下休息一会儿，看着 PIC32 努力为你解决问题。稍过片刻，PIC32 就会运行到断点处，MPLAB 将恢复活跃。然后就能读取 32 位整型变量 t 的数值。本例中 t 的值为 6 140 495！

好了，现在你终于可以理解我为什么建议使用 32 位定时器了。尽管傅里叶变换非常快，但是它的工作量很大，16 位定时器不足以记录如此巨大的时钟周期数。

如果还记得振荡器和主时钟通路的配置，就能很容易地将定时器计数值转换成实际的秒、毫秒和微秒。PIC32 系统的总线时钟频率为 72MHz，而所有外围设备则使用 36MHz 的外围设备总线时钟。将定时器的计数值除以外围设备总线的频率，可得：

$$T = t / F_{pb} = 6\,142\,543 / 36\,000\,000 = 0.17062\text{s}$$

我们还可以让 PIC32 自动完成这个转换，只需在 stopwatch 捕获语句后添加下面这行代码：

$$f = t / 36\text{E}6$$

这将再次使用变量 f 进行浮点除法运算。将 f 添加到 Watch 窗口中，这样就能看到以秒及其分数表示的实验结果（参见图 7-7）。

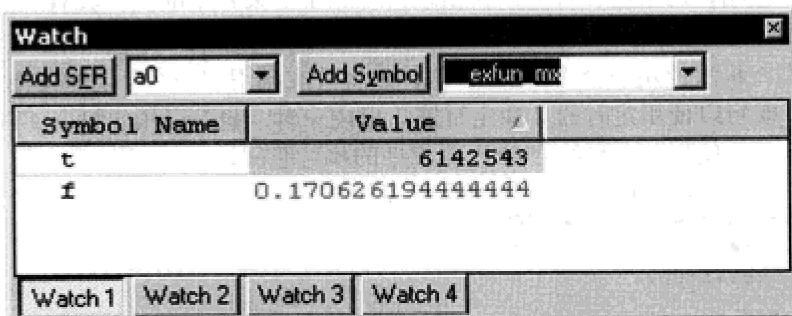


图 7-7　利用 32 位定时器测试 PIC32 的性能

7.11　微调 PIC32：配置 Flash 等待状态

无论你认为用 170ms 完成 256 点 FFT 是否够快，有一点我可以确定：PIC32 还可以做得更好。事实上，除了选择最快的时钟以及合理配置振荡器模块，还需注意微调 PIC32 的其他先进功能，从而获得最佳性能。影响嵌入式控制处理器性能的首要因素是它的 Flash 存储器的速度。但是这里也存在矛盾：Flash 存储器的速度越快，功耗也越大。

PIC32 的设计者发现，使用低功耗 Flash 存储器并将 PIC32 内核系统总线与存储器总线分离可以获得最佳的速度与功耗的平衡。分离的方法是，增加一些等待状态（最多对应 7 个时钟周期），使存储器等待数据从 Flash 存储器中提取出来。根据内核和存储器的速度差别，可能需要增加等待状态的数量。上电时默认选择处于最安全的条件，即等待状态的个数最大。因此，我们就有机会减少它，同时达到符合给定器件实际运行条件的最小值。该等待状态的个数由特殊功能寄存器 CHECON（参见图 7-8）的 PFMWS 位控制。

我们可以直接对 CHECON 寄存器的各位赋值，比如：

```
CHECONbits.PFMWS = 7;    // set max number of waitstates
```

但是，我们必须确定应用系统在最差条件下仍能安全运行（取决于器件数据手册给出的电气特性）所需的最小等待状态的个数。事实上，如果我们使用了错误的等待状态数，那么从 Flash 存储器中执行代码就会出错，而且还会使问题更严重，这种问题只能在特殊的供电电压和温度条件下才能检测出来。

U-0	U-0	U-0	U-0	U-0	U-0	U-0	U-0
—	—	—	—	—	—	—	—
31	30	29	28	27	26	25	24
U-0	U-0	U-0	U-0	U-0	U-0	U-0	R/W-0
—	—	—	—	—	—	—	CHECOH
23	22	21	20	19	18	17	16
U-0	U-0	r-0	r-0	U-0	U-0	R/W-0	R/W-0
—	—	—	—	—	—	DCSZ [1:0]	
15	14	13	12	11	10	9	8
U-0	U-0	R/W-0	R/W-0	U-0	R/W-1	R/W-1	R/W-1
—	—	PREFEN[1:0]		—		PFMWS [2:0]	
7	6	5	4	3	2	1	0

图 7-8　CHECON 控制寄存器

更好的替代方法是，使用 PIC32MX 外围设备函数库提供的 ad hoc 库函数：`SYSTEMConfigWaitStatesAndPB(freq)`。该函数要求以整数参数形式传递系统时钟频率，并且 PIC32 应用支持小组还将它设计成采用针对特定系统时钟频率而“推荐”的最小等待状态，这完全是依靠猜测决定的。



注解　假定函数名的...AndPB 部分可以提醒我们，该函数还能自动修改 PB 分频器的外围设备时钟频率设置，从而保证外围设备总线时钟始终为 50MHz。事实上，这正是我们（在上电时）对系统所做的通用配置。

接下来将在我们的工程上做第二种尝试，即通过添加下面的代码来“调节”等待状态（该代码放在 main() 函数的初始化部分）：

```
SYSTEMConfigWaitStatesAndPB(72000000L);
```

重新生成 Running 工程，并且重新烧录开发板。再次运行程序，直到达到断点（参见图 7-9）。

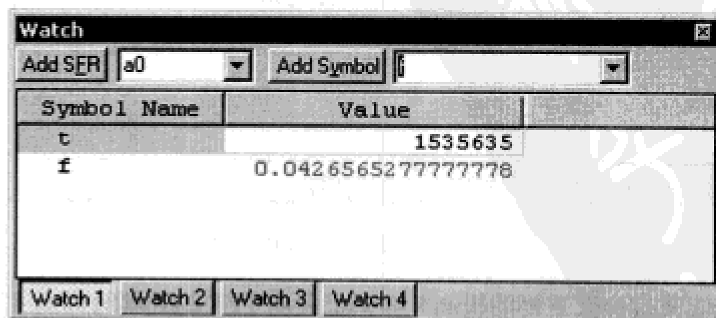


图 7-9　调整等待状态后的 PIC32 性能

看，性能已经有所改善！我们将 FFT 的运行时间由 170ms 降低到了 42ms。性能提高了 4 倍多。

7.12　微调 PIC32：打开指令和数据缓存

我们还有很多事情可以做。随着对 PIC32 架构的进一步了解，我们注意到 MIPS 内核总线

和存储器总线之间还有一个全新的模块：缓存（cache）。它可以视为处理器与 Flash 存储器之间的一种小型而高速的 RAM 存储器块。处理器每次从 Flash 存储器中提取指令或数据时，缓存模块就会保留一个副本，而且还会记住地址。如果处理器（在不久的）将来再次需要（从相同的地址提取）同样的数据，在缓存中就能迅速找到它，从而避免再次访问 Flash 存储器块（并且避免相关的等待状态）。

缓存模块的空间越大，找到某段数据或者指令副本的可能性也越大。反之亦然：算法的内循环越短，缓存对性能的影响也越高。这是因为，一旦所有的缓存都被填满，如果再次提取新指令，那么就必须"替换"缓存内容，最久或者最不常用的指令/数据就会被新信息覆盖。

遗憾的是，由于自身特性的关系，缓存价格昂贵，PIC32MX 的设计者在成本与性能间折中后，设计了 16 行、每行 16B、共 256B 的缓存，它可以保存 64 条完整的 32 位指令。

PIC32 缓存的内部工作机理非常灵活（因此也很复杂），但是我们目前不需要更多的知识也能确定，缓存模块很好，我们当然想使用它。然而事实上，上电时它默认为关闭状态。与前面情况一样，也有一个库函数可以方便地用于控制缓存模块：

```
CheKseg0CacheOn();
```

> **注解**　Kseg0 是 MPLAB C32 在编译工程代码时默认分配给所有代码段的虚拟内存空间。你应该记得位于这个地址空间内的代码"可以"被缓存。而位于 Kseg1 的代码，无论缓存模块是否已经打开，都不能被缓存。

重新生成 Running 工程，并且重新烧录开发板。将应用程序运行到断点处（参见图 7-10）。

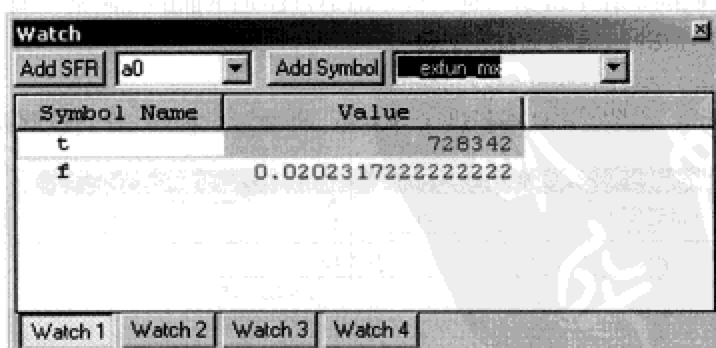


图 7-10　打开缓存后的 PIC32 性能

现在，系统性能又有一次重要的提高！我们已经将 FFT 的处理时间由 42ms 缩短到 20ms。运行速度又加快了 2 倍多。

7.13　微调 PIC32：打开预取指令功能

然而，我们的优化工作还远没有结束。PIC32 的缓存模块一旦被打开，就还有另一个重要功能。它能用于预取指令。也就是说，缓存不仅记录 PIC32 内核正在提取的指令，还能每次"提前运行"并读取一整块的 4 条指令（对应 4 个 32 位的指令字）。如果代码是顺序执行的，那么提取接下来的 3 条指令时所需的等待状态个数都为零。每执行一次跳转指令并打断程序流，预取的缓存数据就会

被丢弃，并载入正确的下一条指令，但是并不会有超出所需等待状态以外的任何其他惩罚。

上电时，缓存的预取指令功能默认处于关闭状态，CHECON 寄存器的 PREFEN 位用于控制该功能。我们可以直接访问该 SFR，也可以利用 pcache.h 中定义的宏 mCheConfigure() 访问它：

```
mCheConfigure(CHECON | 0x30);
```

将这行代码添加到 main() 函数的初始化部分后，重新生成工程，并重新对开发板编程。请将应用程序运行到断点处（参见图 7-11）。

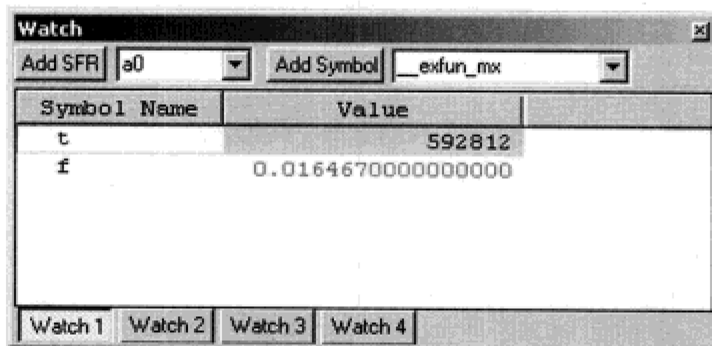


图 7-11 启用缓冲区后的 PIC32 性能

我们又将 FFT 的执行时间从 20ms 减少到 16.4ms，性能又提高了 20%。

7.14 微调 PIC32：最后一步

正如预料的那样，cache 模块其实相当复杂，如果你愿意深究，就会发现大量“技巧”。下面我将介绍最后一个有关访问 RAM 存储器的问题。RAM 存储器的常规访问在默认情况下也会被延误一个等待状态。然而，cache 极大地缓和了这种负面影响，高效的编译器以及使用处理器寄存器也进一步减弱了它对整个处理器性能的影响。尽管如此，仍然有必要使用 mBMXDisableDRMWaitState() 函数来消除这个等待状态。

在本书的实验中，只产生了几乎难以觉察的性能改进，然而根据应用的不同，性能提高的程度也大不相同（参见图 7-12）。

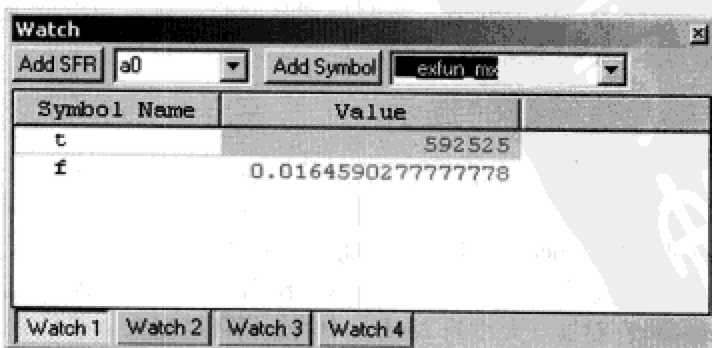


图 7-12 去除 RAM 等待状态后的 PIC32 性能

增加最后这个微调选项后重新生成工程，又获得了一个额外的 1%的性能改善。

总之，与刚开始使用默认配置情况时测量的初始结果相比，仅通过这 4 行代码就能产生几

乎不可想象的性能改善。我们已经将 FFT 算法的运算时间从 170.62ms 降低到 16.45ms，相当于性能提高了 10 倍！

```
// configure PB frequency and the number of wait states
SYSTEMConfigWaitStatesAndPB(72000000L);

// enable the cache for max performance
CheKseg0CacheOn();

// enable instruction prefetch
mCheConfigure(CHECON | 0x30);

// disable RAM wait states
mBMXDisableDRMWaitState();
```

PIC32 单片机的技术支持小组已经为我们准备了简化方式，从现在开始，只需一个简单的库函数就能完成上面所有的优化：

```
SYSTEMConfigperformance(72000000L);
```

这个珍贵的小函数能够微调 Flash 存储器和 RAM 访问，释放出 PIC32 的 cache 和预取模块的威力。将这个函数重新命名为 SportTuning() 或者 RacingMode() 如何？

7.15 小结

在本章中，我们逐步学习了如何调整 PIC32 单片机的引擎。首先是一些粗糙的步骤，然后逐步采用一些精细的步骤，直到我们能够“榨取”出机器的最佳性能。请记住，这个调整过程和当前执行的任务密切相关。不同的应用程序对我们今天调整各种“控制旋钮”的反应也不同。此外，这里获得的结果完全不代表 PIC32 能实现 FFT 的最快速度。事实上，我们谨慎地决定不对原始的算法做任何修改，而只使用了 PIC32MX 架构中的各种硬件功能来提升系统的性能。在该过程中，我们还学到了一些关于外围设备配置特别是 PIC32 定时器模块的新知识，我们可以使用两个 PIC32 定时器模块组成 32 位定时器。

7.16 对汇编语言行家的提示

我们再一次抵制住了手动优化的诱惑，没有使用汇编语言。在实际应用中，那些希望更多了解 PIC32 汇编语言的读者很快就会发现，PIC32 指令集中有一些功能强大的指令，可以进一步提高单片机在信号处理应用中的性能。我要特别强调的是乘累加指令[或者称为乘加指令（MADD）]，它们都是 MIPS 的术语。

7.17 对 PIC 单片机行家的提示

幸亏有 cache 和预取指令功能，即使 PIC32 单片机运行在最高时钟频率下，并使用低功耗 Flash 存储器，它“几乎”也能在每个时钟周期执行一条指令。这里使用的词是“几乎”，因为我们无法确定是否总是如此。cache 会不可避免地时不时出现不命中（misses）；比如，MCU 可能需要一次又一次地等待处理器通过预取指令提取一组命令字，或者将一组新数据载入缓存。完全位于 PIC32 的 cache 存储器（大小为 256B）中的短循环越复杂，不命中的比例就越小。顺便说一句，尽管本书没有足够的时间和篇幅深入讨论这个问题，但是通过缓存的大部分控制寄存器我们已经深入了解了 cache 的工作过程，并了解了某段代码的“轮廓”。

因此，能否将 PIC32 称作 72MIP5 机呢？（这意味着它真能在 1 秒中执行 7200 万条指令。）我认为明智的答案是“大部分情况下都可以这么叫”，但是，这取决于代码以及如何使用缓存。

7.18 提示与技巧

MPLAB IDE 自带的另一个强大工具是 Data Monitor（数据监视器）与 Control Interface（控制接口），或者简称为 DCMI。在 MPLAB IDE 的主菜单中选择 Tools|DCMI 就可以激活它。当与任何一款在线调试器甚至 MPLAB SIM 仿真器联合使用时，它能够通过图形为我们提供一个了解器件数据空间的窗口，并允许我们通过一组可配置图形化用户接口（GUI）“交互地”修改数据。特别是当处理 FFT 时，你可能想检查合成的（正弦）输入信号的形状，并希望形象地查看 FFT 程序的输出。打开 DCMI 窗口后，请按照顺序执行下列步骤。

（1）单击 Dynamic Data View 选项卡。

（2）选择 Graph1 选项框。

（3）在 first graph 上单击，激活上下文菜单。

（4）选择 Configure Data Source（参见图 7-13）。

（5）在 Global Symbols（全局符号列表）中选择 inB 缓存。

（6）单击 OK 按钮。

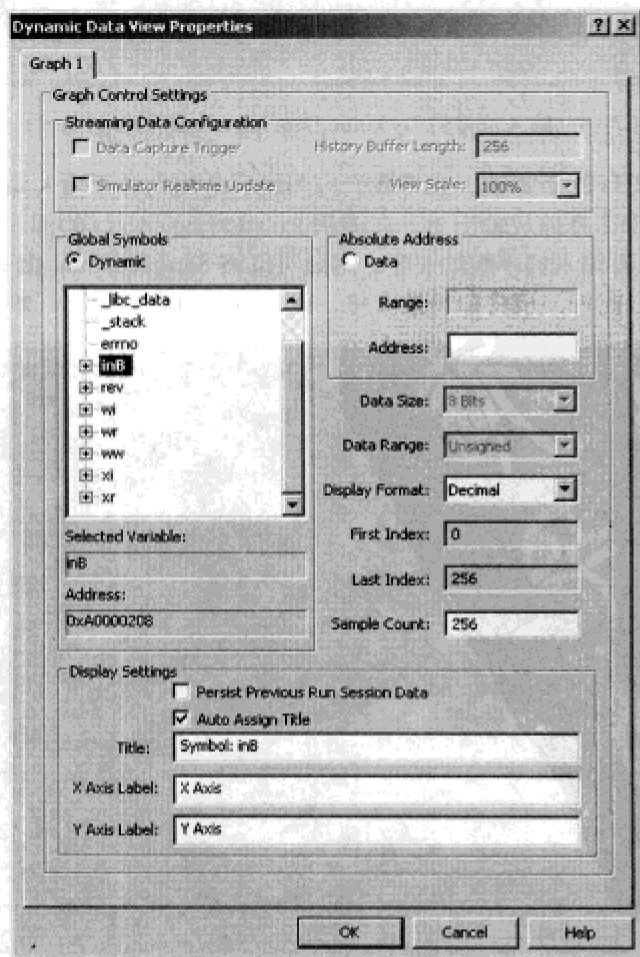


图 7-13　DCMI 的 Dynamic Data View Properties 对话框

下面请在 inB[]缓存初始化后调用 OpenTimer45()的那一行设置断点，然后运行程序。

当程序停止时，就能在 Dynamic Data View（动态数据观察）窗口（参见图 7-14）中看到 inB[]缓存的内容。

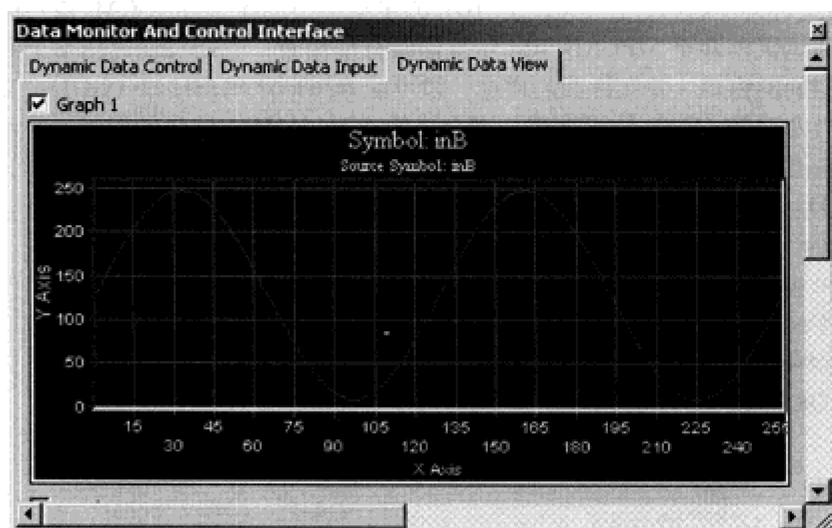


图 7-14　输入信号的 Dynamic Data View（动态数据视图）

这是一个频率为 2Hz 的正弦波，或者我们应当说它是周期等于输入采样数一半的正弦波。

现在可以在完成 FFT 并缩放输出后，在调用 ReadTimer45()的那一行设置第二个断点。请记住，FFT 的输出只有输入样本数的一半，因此可以将 Sample Count 由 DCMI 自动设定的默认值（256）改为 128。此外，请将窗口最大化，以便更好地查看细节（参见图 7-15）。

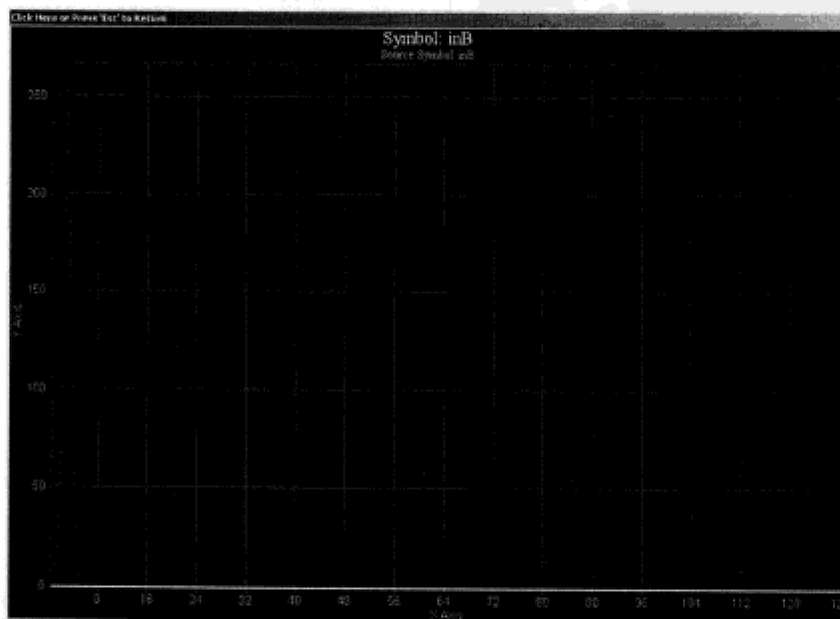


图 7-15　FFT 输出的 Dynamic Data View（动态数据视图）：信号的频谱

可见，信号功率谱的唯一的尖峰位于横轴上（假设样本数从1开始计数）的2Hz处（或者等于输入样本数内的两个周期）。这与我们设计的输入测试信号是完全一致的！

7.19 练习

（1）在进行功率缩放前，请验证FFT输出的形状和大小（实部和虚部）。
（2）去掉窗口，观察信号的频谱是否会变化以及如何变化。
（3）利用多输入正弦波创建一个复合信号并查看其FFT输出。
（4）为输入空间分配更多空间，然后实验，观察结果的性能变化。

7.20 参考书

Dominic Sweetman所著的《MIPS体系结构透视（第2版）》（*MIPS RUN*）。如果你想真正了解PIC32 MIPS内核的先进功能，就一定要阅读这本书。推荐第2版的原因是，它更注重MIPS内核的现代实现方法，并且增加了在Linux内核实现MIPS的详细说明。（请不要在家里对PIC32MX做上述尝试，至少目前还不要这样做）。

7.21 链接

http://en.wikipedia.org/wiki/FFT。该网站有助于学习更多有关快速傅里叶变化的用途和实现方法的内容。

http://en.wikipedia.org/Spectral_music。FFT也可以非常有趣！想想图像和音乐的合成。

http://en.wikipedia.org/Window_function。注意，这里讨论的并不是实际的窗户，然而这些窗口能够戏剧性地改变你的视角！

http://en.wikipedia.org/CPU_cache。PIC32MX是第一款使用缓存的PIC单片机，因此，有必要深入研究该主题，并了解PIC32的设计师们在保持产品低价格的同时，为了获得最佳的性能都做了哪些设计与折中。

第8章 通 信

8.1 计划

除了最简单的嵌入式控制应用系统以外，大部分嵌入式应用系统都需要和其他智能化设备通信。这些设备可能是个人电脑、传感器、显示器，或者是同一电路板上的或远程的其他单片机。为了降低成本，所采用的解决方案通常只能包含少量引脚和连线，因而导致用户考虑选择串行通信接口。

在嵌入式控制应用系统中，设计通信的过程就是理解协议以及物理介质特性的过程。学会为系统选择合适的通信接口与掌握其用法同样重要。

本章将比较 PIC32MX 系列所有通用型单片机都支持的各种基本通信接口，着重介绍异步串行通信接口（UART）和同步串行通信接口（SPI 和 I^2C），比较它们在嵌入式控制应用系统中的通信距离和使用限制。

8.2 准备

本章除了要使用 MPLAB IDE、MPLAB C32 学生版编译器以及 MPLAB SIM 仿真器等软件外，还需要 Explorer 16 演示板和一台在线调试器，比如 MPLAB ICD2、MPLAB ICD3、MPLAB REAL ICE 或 PIC32 Starter Kit。如果打算使用 PIC32 Starter Kit，那就还需要专用的 PIC32 Starter Kit 适配器（PIM）。

8.3 探索

PIC32MX 系列单片机支持 7 种通信外围设备，可用于各种嵌入式控制应用系统。其中有 6 种是串行通信外围设备，每次只收发一字节的信息：

- ❑ 2 个通用异步收发器（UART）；
- ❑ 2 个 SPI 同步串行接口；
- ❑ 2 个 I^2C 同步串行接口。

同步通信接口（比如 SPI 或 I^2C）和异步通信接口（比如 UART）的区别主要在于时序信息在发送器和接收器间的传送方式。同步通信外围设备需要一个用于传输时钟信号的物理连接线（电线），实现两个设备的同步。提供时钟信号的设备通常被称为主设备，而另一个与之同步的设备则称为从设备。

8.4 同步串行接口

如图 8-1 所示，I^2C 接口有 2 根线，因此需要使用单片机的 2 个引脚：一个作为时钟信号（SCL），另一个作为双向数据信号（SDA）。

SPI 接口则有两根独立的数据线（参见图 8-2），其中一个用于输入（SDI），另一个用于输出（SDO），尽管需要的连线更多，但是能够实现更快速的双向同步数据传输。

为了使同一个串行通信接口（同一总线）连接多个设备，I^2C 接口要求在传输数据前向数据总线上发送 10 位地址。这会减缓通信速率，但是能够在 2 根连线（SCL 和 SDI）上实现多达 1000 台设备的通信（理论值）。此外，I^2C 接口还能通过简单的仲裁协议实现多个主设备对总线

的共享。

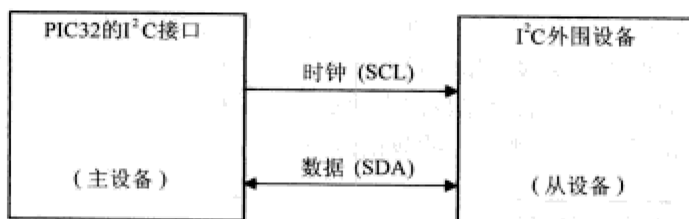


图 8-1 I²C 接口的框图

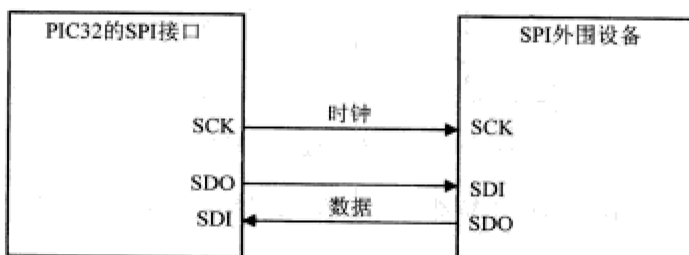


图 8-2 SPI 接口的框图

另一方面，如图 8-3 所示，SPI 接口还需要一根物理连线[从设备选择（SS）]连接每个设备。这意味着在实际使用 SPI 总线时，随着连接设备数量的增加，所需 PIC32 的 I/O 引脚数也随之成比例增加。

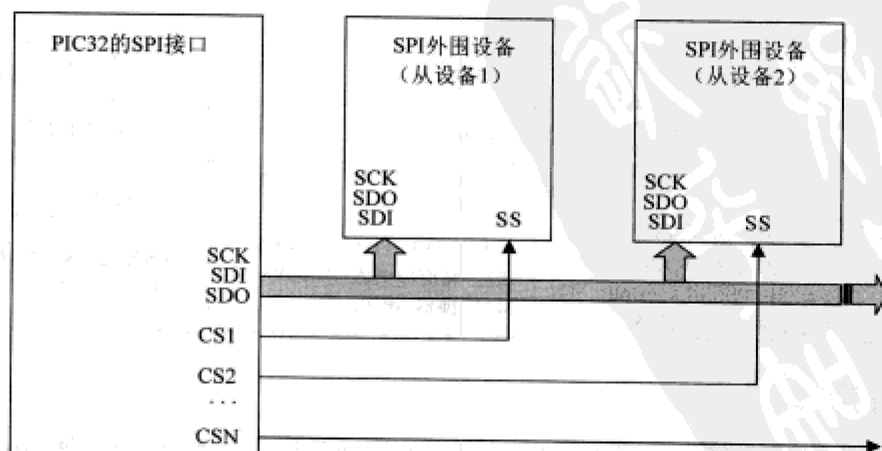


图 8-3 SPI 总线的框图

理论上允许同时有多个主设备共享 SPI 总线，但实际中很少这样使用。SPI 接口的优点是简单，并且传输速率比最快的 I²C 总线还高一个量级（即使不考虑由协议本身带来的时间开销也是如此）。

8.5　异步串行接口

如图 8-4 所示，异步串行接口没有时钟线，它通常只有 2 根数据线 TX 和 RX，用作输入和输出。此外还有两根可选的连接线，用作硬件握手。发送器与接收器的同步是通过从数据流本身提取时序信息实现的。为此，需要在数据中增加起始位和终止位，并且需要设置精确的数据格式（以及固定的波特率），以实现可靠的数据传输。

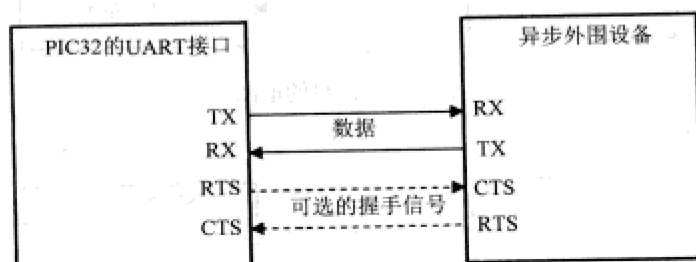


图 8-4　异步串行接口的框图

有些异步串行接口协议要求使用特殊的收发器，以提高抗噪性，并将物理连接距离增加到数千英尺[①]。

每种串行通信接口都有其优缺点。表 8-1 总结了其最重要的优缺点以及最常见的应用。

表 8-1　串行接口的对比

外围设备	异　步		同　步
	SPI	I^2C	UART
最大位速率	20Mbit/s	1Mbit/s	500kbit/s
最大总线数	受引脚数限制	128 个	点到点（RS232），256 个（RS485）
需要的引脚数	$3+n\times$CS	2	2（+2）
优点	简单、成本低、速度高	引脚数少、允许多个主设备	传输距离更远（使用收发器提高抗噪性）
缺点	只允许一个主设备、传输距离短	最慢、传输距离短	需要精确的时钟频率
常见应用	在同一块 PCB 板上直接连接多个外围设备	在同一块 PCB 板上经总线连接多个外围设备	与终端、个人电脑及其他数据采集系统接口
应用示例	串行 EEPROM（25CXXX 系列），AD 转换器 MCP320X，以太网控制器 ENC28J60，CAN 总线控制器 MCP251X	串行 EEPROM（24CXXX 系列），温度传感器 MCP98XX，AD 转换器 MCP322X	RS232，RS422，RS485，LIN 总线，红外接口 MCP2552

8.6　并行接口

并行主接口（Parallel Master Port，PMP）是 PIC32 单片机首次增加的基本通信接口。PMP 一次能够传输 16 位数据，并提供能与很多商用型 LCD 显示屏（带有集成控制器的字符式和图形式模块）、CF（压缩闪存）卡（或者 CF-I/O 卡）、打印机以及市面上几乎所有其他基本的 8 位和 16 位并行设备直接相连的地址线，以及标准控制信号-CS、-RD 和-WR。

① 1 英尺≈30.48 厘米。——编者注

本章只介绍同步串行端口 SPI 的使用。在接下来的几章里则将讨论异步串行接口和 PMP 接口。

8.7 基于 SPI 的同步通信

尽管 PIC32 单片机的 SPI 接口具有很多功能选项和有趣的特性，但是它可能是其中最简单的外围设备了。

如图 8-5 所示，SPI 接口实际上是一个移位寄存器。在 SCK 引脚上时钟信号的同步下，按照最高位（MSB）先进的原则，数据位被同步地从 SDI 脚移入，又从 SDO 脚移出。移位寄存器的大小可以是 8 位、16 位或者 32 位。

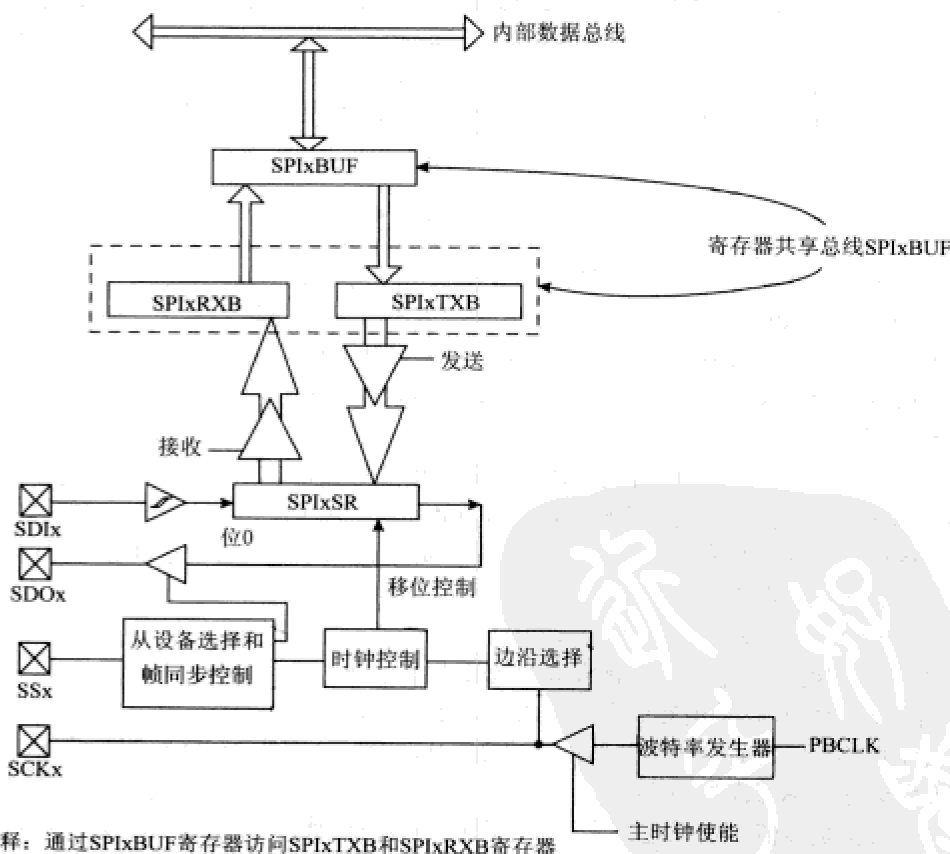


图 8-5 SPI 模块的组成框图

如果把设备配置为总线主设备，那么时钟信号将在设备内部由外围设备总线时钟（F_{pb}）经过波特率发生器产生，并输出在 SCK 引脚上。另一方面，该设备是总线从设备，它从 SCK 引脚上接收时钟信号。

和我们接触过的其他外围设备一样，SPI 中关键的配置选项是由特殊功能寄存器 SPIxCON 以及波特率产生控制寄存器 SPIxBRG 控制的（参见图 8-6）。

请注意，在图 8-6 中，SPIxCON 寄存器的低 16 位（最低有效位）包含所有的关键配置位，而高 16 位则只控制 SPI 接口的高级功能（帧模式）。这使得 SPIxCON 控制寄存器能与过去的

16位PIC单片机兼容（它们的高位默认都为零）。

R/W-0	R/W-0	R/W-0	U-0	U-0	U-0	U-0	U-0
FRMEN	FRMSYNC	FRMPOL	—	—	—	—	—
位31							位24

U-0	U-0	U-0	U-0	U-0	U-0	R/W-0	U-0
—	—	—	—	—	—	SPIFE	—
位23							位16

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
ON	FRZ	SIDL	DISSDO	MODE32	MODE16	SMP	CKE
位15							位8

R/W-0	R/W-0	R/W-0	U-0	U-0	U-0	U-0	U-0
SSEN	CKP	MSTEN	—	—	—	—	—
位7							位0

图8-6 控制寄存器SPIxCON

为了演示SPI外围设备的基本功能，我们将使用Explorer 16演示板，其中PIC32单片机的SPI2模块与串行EEPROM芯片（可记为SEE或者E^2，念作"E方"）25LC256相连。这是一片价格便宜的小尺寸非易失型长寿命存储器，包含256k bit（32KB）。

请使用New Project Setup(创建新工程)检查表创建一个名为SPI的工程,并创建名为spi2.c的源文件。

配置SPI2模块与串行存储器设备通信的最直接的方法是，手动为SPI2CON寄存器的各位赋值。根据25LC256芯片的数据手册(编号为DS21822，可以从Microchip公司的网站上下载)，该串行存储器通过SPI接口接受8位命令字时，必须遵照下述设置（请注意括号内SPI2CON寄存器的控制位的对应值）：

- ❑ 8位模式（MODE16=0，MODE32=0）；
- ❑ 时钟空闲时为低电平，时钟有效时为高电平（CKP=0）；
- ❑ 串行输出值在时钟由有效变向空闲时改变（CKE=1）。

还需要将PIC32配置成SPI总线的主设备（MSTEN=1），因此存储器是从设备，换句话说，它需要接收来自SCK引脚的时钟信号。

上述配置位是数值，可以定义成常数，以便稍后一起赋值给SPI2CON寄存器：

```
// peripheral configurations

#define SPI_CONF 0x8120 // SPI on, 8-bit master, CKE=1,CKP=0
```

为了设定波特率，需要使用公式（8-1）（取自PIC32单片机的数据手册）。

公式（8-1）确定SPI时钟频率的公式

$$F_{sck} = \frac{F_{pb}}{2\times(\text{SPIxBRG}+1)}$$

既可以使用SPI2BRG寄存器的默认值（上电时为0，对应的分频系数为1:2），也可以设定一个合适的值以减慢通信速度，这有助于降低EEPROM的功耗，例如：

```
#define SPI_BAUD  15  // clock divider Fpb/(2 * (15+1))
```

在该设置条件下，波特率是 F_{pb} 的 1/32。如果在程序的最开始添加下列几行代码，将 PIC32 的外围设备总线时钟设为 9MHz，那么对应的波特率就是 280kHz。

```
// configuration bit settings, Fcy=72MHz, Fpb=9MHz
#pragma config POSCMOD=XT, FNOSC=PRIPLL
#pragma config FPLLIDIV=DIV_2, FPLLMUL=MUL_18, FPLLODIV=DIV_1
#pragma config FPBDIV=DIV_8, FWDTEN=OFF, CP=OFF, BWP=OFF
```

从 Explorer 16 用户手册中（DS51589 的附录 A 电路板原理图），我们可以了解到 PortD 的引脚 12 与存储器的片选引脚（CS）相连。请注意，该引脚是低电平有效。增加以下两行定义可以增强代码可读性。

```
// I/O definitions
#define CSEE        _RD12        // select line for EEPROM
#define TCSEE       _TRISD12     // tris control for CSEE pin
```

下面将编写范例程序的外围设备初始化代码：

```
// 1.init the SPI peripheral
TCSEE = 0;                  // make SSEE pin output
CSEE = 1;                   // de-select the EEPROM
SPI2CON = SPI_CONF;         // select mode and enable
SPI2SPI2BRG = SPI_BAUD;     // select clock speed
```

然后编写从串行 EEPROM 收发数据的函数：

```
// send one byte of data and receive one back at the same time
int writeSPI2( int i)
{
  SPI2BUF = i;                    // write to buffer for TX
  while( !SPI2STATbits.SPIRBF);   // wait for transfer complete
  return SPI2BUF;                 // read the received value
}//writeSPI2
```

writeSPI2()实际上是一个双向传输函数。它能立即向发送缓冲区发送一个字符，然后进入一个循环，等待接收标志置位，表示已完成发送，并从设备接收到数据。最后将接收到的数据作为函数返回值传回。

然而，在与存储器通信时，存在向存储器发送命令后没有立即响应的情况，也存在从存储器读取数据后，PIC32 无需再发出命令的情况。对于第一种情况（比如，写命令），可以忽略函数的返回值。对于第二种情况（比如，读命令），可以在读取的同时向存储器发送一个伪数据。

25LC256 的数据手册包含用于从存储器读写数据的 7 种命令时序的准确描述。利用以下常数表可以帮助我们在代码中对这些命令编码：

```
// 25LC256 Serial EEPROM commands
#define SEE_WRSR    1        // write status register
#define SEE_WRITE   2        // write command
#define SEE_READ    3        // read command
#define SEE_WDI     4        // write disable
#define SEE_STAT    5        // read status register
#define SEE_WEN     6        // write enable
```

在尝试其他更复杂的任务前，首先测试一下我们已经编写的代码段，验证它能否与设备正常通信。例如，可以使用读状态寄存器（SEE_STAT）命令查询 EEPROM 的状态，并获取内部状态寄存器的值。

8.8　测试读状态寄存器命令

首次调用 writeSPI2()函数发送正确的命令字（SEE_STAT）后，还需要发送第一个（无用的）数据，以便捕获存储器器件的响应（参见图 8-7）。

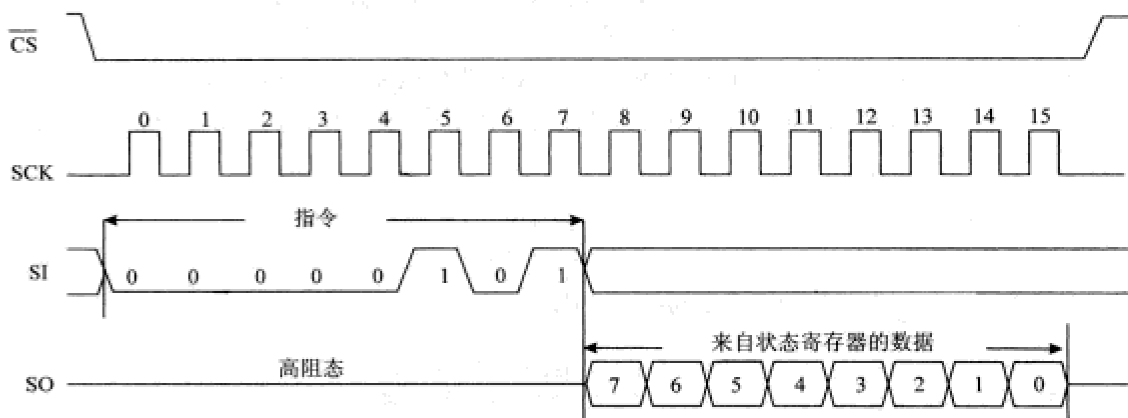


图 8-7　完整的读状态寄存器命令的时序

无论向串行 EEPROM 发送什么命令，都至少需要执行以下步骤。

(1) 将 CS 引脚置低，激活存储器。

(2) 逐位移出 8 位命令字。

(3) 根据具体的命令发送或接收多字节数据。

(4) 释放存储器（将 CS 引脚置高）以完成命令。完成这一步后，存储器会返回到低功耗待机模式。

在实际应用中，需要以下代码实现完整的读状态寄存器操作：

```
// Check the Serial EEPROM status
CSEE = 0;                       // select the Serial EEPROM
writeSPI2( SEE_STAT);           // send a READ STATUS COMMAND
i = writeSPI2( 0);              // send dummy, read data
CSEE = 1;                       // deselect to complete command
```

完整的工程代码如下：

```
/*
** SPI2
**
*/
#include <p32xxxx.h>

// configuration bit settings, Fcy=72MHz, Fpb=9MHz
#pragma config POSCMOD=XT, FNOSC=PRIPLL
#pragma config FPLLIDIV=DIV_2, FPLLMUL=MUL_18, FPLLODIV=DIV_1
#pragma config FPBDIV=DIV_8, FWDTEN=OFF, CP=OFF, BWP=OFF

// I/O definitions
#define CSEE  _RD12             // select line for Serial EEPROM
#define TCSEE _TRISD12          // tris control for CSEE pin

// peripheral configurations
#define SPI_CONF   0x8120       // SPI on, 8-bit master,CKE=1,CKP=0
#define SPI_BAUD   15           // clock divider Fpb/(2 * (15+1))
```

```
// 25LC256 Serial EEPROM commands
#define SEE_WRSR    1       // write status register
#define SEE_WRITE   2       // write command
#define SEE_READ    3       // read command
#define SEE_WDI     4       // write disable
#define SEE_STAT    5       // read status register
#define SEE_WEN     6       // write enable

// send one byte of data and receive one back at the same time
int writeSPI2( int i)
{
    SPI2BUF = i;                    // write to buffer for TX
    while( !SPI2STATbits.SPIRBF);   // wait for transfer complete
    return SPI2BUF;                 // read the received value
}//writeSPI2

main ()
{
  int i;
  // 1. init the SPI peripheral
  TCSEE = 0;                        // make SSEE pin output
  CSEE = 1;                         // de-select the Serial EEPROM
  SPI2CON = SPI_CONF;               // select mode and enable SPI2
  SPI2BRG = SPI_BAUD;               // select clock speed
  // main loop
  while( 1)
  {
    // 2. Check the Serial EEPROM status
      CSEE = 0;                     // select the Serial EEPROM
      writeSPI2( SEE_STAT);         // send a READ STATUS COMMAND
      i=writeSPI2( 0);              // send/receive
      CSEE=1;                       // deselect terminate command
  } // main loop
} // main
```

根据你选用的调试工具，选择合适的 Debugger Setup（调试器设置）检查表，打开在线调试功能，并准备配置工程。根据 Project Build（生成工程）检查表编译和链接代码后，执行以下操作。

(1) 连接 Explorer 16 演示板，然后选择 Debugger|Program 选项烧录 PIC32 单片机。MPLAB 会默认使用最小的存储空间将工程代码传输到器件，使烧录时间最短。大约几秒钟后，PIC32 单片机就被烧录、校验完毕，可以准备运行了。

(2) 在工程中添加 Watch（观察）窗口。

(3) 在符号选择框中选择 i，然后单击 Add Symbol 按钮。

(4) 将光标指向主循环中的最后一行代码（包括撤消 CSEE），然后设置断点（双击即可）。

(5) 选择 Debugger|Run 命令，开始执行。

(6) 当执行结束时，25LC256 存储器状态寄存器的内容传入变量 *i*，并能从 Watch 窗口中看到。

遗憾的是，你可能会失望地发现 25L256 存储器（上电时）的默认状态是用 0×00 的形式来表示数据的（参见表 8-2）。

表 8-2 是 EEPROM 状态寄存器的内容，事实上，从该表以及器件的数据手册中可以看出，除非已经设置了块代码保护，否则在上电时块保护位（BP1 和 BP0）就应该清零，这样写使能锁存（Write Enable Latch，WEL）被禁止，并且不会激活 WIP（Write In Progress）标志。

表 8-2　25LC256 串行 EEPROM 的状态寄存器

7	6	5	4	3	2	1	0
W/R	–	–	–	W/R	W/R	R	R
WPEN	×	×	×	BP1	BP0	WEL	WIP

W/R=可写/可读，R=只读

这个小测试程序并不是很有说服力。为了使测试更有意思，可以在查询状态寄存器前置位写使能锁存位；寄存器的第 1 位置位后就好了。

请将下述代码插入第二节最前面，而以前的代码则被重新编号为 2.2：

```
// 2.1 send a Write Enable command
CSEE = 0;                    // select the Serial EEPROM
writeSPI2( SEE_WEN);        // send command, ignore data
CSEE=1;
```

（1）重新生成工程。

（2）重新烧录器件。

（3）运行（或者运行到光标处）。

如果一切正常，就会看到 Watch 窗口里的变量 *i* 变红，并显示数值为 2。只有在对强大的 32 位嵌入式控制器编程时才能获得如此巨大的满足感！

再认真地想一下，既然写使能锁存位已被置位，那么就能增加一条写命令，开始“修改”EEPROM 器件的内容。可以一次写一字节，或者可以在一次命令（称为页写入命令）中写一个长字符串，最大可达 64 字节。请仔细阅读数据手册中有关执行该命令时在地址限制方面的内容。

8.9　向 EEPROM 写数据

发送写命令后，必须在发送实际数据前先发送两字节的地址。以下代码示范了正确的写顺序：

```
// send a Write command
CSEE = 0;                    // select the Serial EEPROM
writeSPI2( SEE_WRITE);       // send command, ignore data
writeSPI2( ADDR_MSB);        // send MSB of memory address
writeSPI2( ADDR_LSB);        // send LSB of memory address
writeSPI2( data);            // send the actual data
// send more data here to perform a page write
CSEE = 1;                    // start actual EEPROM write cycle
```

请注意实际的 EEPROM 写周期是在 CS 线再次变高后开始的。此外，在开始新命令前还必须等待一段时间（T_{wc}），以便完成上一个命令周期，时间长短由存储器数据手册指定。

有两种方法可以确定存储器允许的写命令完成时间。最简单的方法是在写操作后插入一段固定的延时。延时的长度要大于存储器数据手册指定的最长写周期时间（例如最大 T_{wc}=5ms）。

更好的方法是在启动下一次读/写命令前检查状态寄存器，并等待 `WIP` 标志清零，它会与写使能（Write Enable，`WEN`）位的复位同步。通过这种方法，只需等待存储器器件在当前运行条件下所需的最小处理时间。

8.10　读取存储器的内容

读取存储器的内容其实更简单。以下代码段就能完成该功能：

```
// send a Write command
CSEE = 0;                 // select the Serial EEPROM
writeSPI2( SEE_READ);     // send command, ignore data
writeSPI2( ADDR_MSB);     // send MSB of memory address
writeSPI2( ADDR_LSB);     // send LSB of memory address
data=writeSPI2( 0);       // send dummy, read data
// read more data here sequentially incrementing the address
CSEE = 1;                 // terminate the read sequence
                          // and return to low power
```

其中，读操作可以重复任意次。如果有必要，甚至可以连续读取整个存储器的内容，直至到达最后一个存储器地址（0x7FFF）后，又重新从0x0000单元开始读取。

8.11 32位串行EEPROM存储器的函数库

现在，可以组建一个专用于访问25LC256串行EEPROM的小型函数库。该函数库可以隐藏所有的实现细节，比如所用的SPI接口、特殊的操作顺序以及详细的时序。它只显示两个基本的命令就能实现对通用（黑盒）非易失型存储器进行整型数据（32位）的读写。

我们将利用Project Wizard（创建工程向导）以及常用的检查表创建一个新工程，并为工程取一个合适的名字SEE。在创建名为see.c的源文件后，就可以复制SPI工程里的大部分定义：

```
/*
** SEE Access Library
*/

#include "p32xxxx.h"
#include "see.h"

// I/O definitions
#define CSEE    _RD12          // select line for Serial EEPROM
#define TCSEE   _TRISD12       // tris control for CSEE pin

// peripheral configurations
#define SPI_CONF  0x8120       // SPI on, 8-bit master,CKE=1,CKP=0
#define SPI_BAUD  15           // clock divider Fpb/(2 * (15+1))

// 25LC256 Serial EEPROM commands
#define SEE_WRSR    1          // write status register
#define SEE_WRITE   2          // write command
#define SEE_READ    3          // read command
#define SEE_WDI     4          // write disable
#define SEE_STAT    5          // read status register
#define SEE_WEN     6          // write enable
```

然后，再复制初始化代码、写函数以及读取状态寄存器指令。它们每个都是独立的函数。

```
// send one byte of data and receive one back at the same time
int writeSPI2( int i)
{
  SPI2BUF = i;                    // write to buffer for TX
  while( !SPI2STATbits.SPIRBF);   // wait for transfer complete
  return SPI2BUF;                 // read the received value
}// writeSPI2

void initSEE( void)
{
  // init the SPI2 peripheral
  CSEE = 1;                  // de-select the Serial EEPROM
  TCSEE = 0;                 // make SSEE pin output
```

```
    SPI2CON = SPI_CONF;              // enable the peripheral
    SPI2BRG = SPI_BAUD;              // select clock speed
}// initSEE

int readStatus( void)
{
    // Check the Serial EEPROM status register
    int i;
    CSEE = 0;                        // select the Serial EEPROM
    writeSPI2( SEE_STAT);            // send a READ STATUS COMMAND
    i = writeSPI2( 0);               // send/receive
    CSEE = 1;                        // deselect terminate command
    return i;
} // readStatus
```

为了编写一个能从非易失型存储器中读取整数的函数，首先要通过读取状态寄存器来验证前一条（写）指令已经正常结束。这里通过连续读取2字节来组成一个整数：

```
int readSEE( int address)
{ // read a 32-bit value starting at an even address

    int i;

    // wait until any work in progress is completed
    while ( readStatus() & 0x1); // check WIP

    // perform a 16-bit read sequence (two byte sequential read)
    CSEE = 0;                        // select the Serial EEPROM
    writeSPI2( SEE_READ);            // read command
    writeSPI2( address >>8);         // address MSB first
    writeSPI2( address & 0xfc);      // address LSB (word aligned)
    i = writeSPI2( 0);               // send dummy, read msb
    i = (i<<8)+ writeSPI2( 0);       // send dummy, read lsb
    i = (i<<8)+ writeSPI2( 0);       // send dummy, read lsb
    i = (i<<8)+ writeSPI2( 0);       // send dummy, read lsb
    CSEE = 1;
    return ( i);
}// readSEE
```

最后，从前面的工程中提取出用于访问写使能锁存的代码段，并增加页面写操作，就构成了写使能函数。

```
void writeEnable( void)
  {
    // send a Write Enable command
    CSEE = 0;                    // select the Serial EEPROM
    writeSPI2( SEE_WEN);         // write enable command
    CSEE = 1;                    // deselect to complete the command
}// writeEnable

void writeSEE( int address, int data)
{ // write a 32-bit value starting at an even address

    // wait until any work in progress is completed

    while ( readStatus() & 0x1) // check the WIP flag

    // Set the Write Enable Latch
    writeEnable ();
```

```
    // perform a 32-bit write sequence (4 byte page write)
    CSEE = 0;                      // select the Serial EEPROM
    writeSPI2( SEE_WRITE);         // write command
    writeSPI2( address>>8);        // address MSB first
    writeSPI2( address & 0xfc);  // address LSB (word aligned)
    writeSPI2( data >>24);         // send msb
    writeSPI2( data >>16);         // send msb
    writeSPI2( data >>8);          // send msb
    writeSPI2( data);              // send lsb
    CSEE = 1;
}// writeSEE
```

这里还可以添加更多的函数，比如访问 short 型（16 位）或者 long long 型（64 位）整数的函数，但我们只是为了验证方法的正确性，因此这样已经足够了。

请注意页面写操作（详细内容请参阅 25LC256 存储器的数据手册）要求地址与两个边界的幂次对齐（这里，只要地址能被 4 整除就行）。为了保持一致，读函数也必须遵守这一要求。

将上述代码保存在 see.c 中，并将它添加到工程中。可以采用检查表中列出的 3 种方法将文件添加至工程。既可以使用编辑器的上下文菜单并选择 Add to Project 命令，也可以在工程窗口中的源文件分支上单击并选择 Add Files，然后从当前工程的目录选择 see.c 文件。

为了使本模块的一些函数能被其他应用程序调用，需要创建一个名为 see.h 的文件，并写入以下声明：

```
/*
** SEE Access library
**
** encapsulates 25LC256 Serial EEPROM
** as a NVM storage device for PIC32 + Explorer16 applications
*/

// initialize access to memory device
void initSEE(void);

// 32-bit integer read and write functions
// NOTE: address must be an even value between 0x0000 and 0x3ffc
// (see page write restrictions on the device datasheet)
int  readSEE ( int address);
void writeSEE( int address, int data);
```

这里只显示了初始化函数和整数读/写函数，而隐藏了所有的实现细节。

在工程窗口的头文件图标上单击，从当前工程目录中选择 see.h 文件，并将它添加进工程。

8.12 测试新的串行 EEPROM 存储器函数库

为了测试新函数库的功能，我们将创建一个测试程序，它包括一些反复读取某个存储器地址（地址 16）的内容的代码，递增所读取的数据值，然后再将结果存回存储器。

```
/*
** SEE Library test
*/
// configuration bit settings, Fcy=72MHz, Fpb=9MHz
#pragma config POSCMOD=XT, FNOSC=PRIPLL
#pragma config FPLLIDIV=DIV_2, FPLLMUL=MUL_18, FPLLODIV=DIV_1
#pragma config FPBDIV=DIV_8, FWDTEN=OFF, CP=OFF, BWP=OFF
#include <p32xxxx.h>
#include "see.h"
```

```
main ()
{
  int data;

  // initialize the SPI2 port and CS to access the 25LC256
  initSEE();
  // main loop
  while ( 1)
  {
    // read current content of memory location
    data = readSEE( 16);
    // increment current value
    data++;      // <-set brkpt here

    // write back the new value
    writeSEE( 16, data);
    //address++;

  } // main loop
} //main
```

(1) 将上述代码保存为文件 SEEtest.c，并将该文件添加到当前的工程中。

调用 Build All 命令，就会看到 MPLAB C32 编译器会依次处理两个源文件（.c），然后将目标代码链接成一个可执行文件（.hex）。

(2) 在 Watch（观察）窗口中增加 data 变量。

(3) 在紧接着的读命令处设置一个断点，测试 SEE 函数库是否运行正常。

(4) 单击 Run 命令，等待程序执行完第一次读操作后停止。

(5) 注意 data 的数值，然后再次单击 Run 命令。变量 data 的数据会不断增加，即使对程序复位，或者将电路板完全断电后过一会儿再重新接好，情况也一样。我们将看到 16 号地址单元的数据保存正常，并且能够成功递增。

当心，如果主程序循环在没有断点的情况下任意运行，那么测试程序会很快测试出串行 EEPROM 的寿命。事实上，该循环会以满足器件实际 T_{wc} 要求的最快速率反复烧写第 16 号单元。在最好的情况下（最大的 T_{wc}=5ms），每秒也能烧写 200 次。换句话说，以 EEPROM 的理论寿命（可重复烧写 1 000 000 次）来算，那么最多也就只能坚持 5000 秒，也就是说只能连续执行不足一个半小时。

8.13 小结

本章开始研究串行接口，比较了各种串行接口的基本差别，并回顾了嵌入式控制系统中最常用的一些串行接口。考虑到 SPI 模块的配置最为简单，我们还进行了通过 SPI 模块访问串行 EEPROM 存储器 25LC256（它是嵌入式应用系统中最常用的非易失性储存器之一）的试验。我们开发的小型函数库可能在今后的应用系统中再次用到，这样就能在 Explore 16 演示板上获得额外的非易失性存储器（32KB）。

8.14 对 C 语言编程行家的提示

习惯于为大型工作站和个人电脑开发代码的 C 语言程序员可能会进一步开发函数库，使其包含最灵活和最全面的函数集。我的建议是，在开始为库函数添加任何新参数前，最好先屏住呼吸，然后数到 10。在嵌入式控制领域，传输的参数越多，就意味着需要使用的栈空间越大，为栈复制数据和从栈中复制数据所花费的时间也越多，并且产生的代码通常也更长。函数库越

简单，就越容易测试和维护。这并不意味着不遵守面向对象的编程习惯，相反，我们的示例程序正好可看作是对象包装的范例，因为我们已经向用户隐藏了 SPI 接口和串行 EEPROM 的内部工作细节，而只向用户提供与按 32 位字进行组织的通用存储器的简单接口。

8.15 对 Explorer 16 专家的提示

Explorer 16 演示板最不为人所知的功能之一，就是板上的两片数字式多路复用器（74HCT4053）U6 和 U7。其中，U6 用于交换 SPI1 端口的 SDI 和 SDO 线，以便在连接 PICTail 连接器时能互联两块 Explorer 16 板，从而使两个单片机能够交换数据。该信号交换功能由 RB12 引脚控制，它被配置为数字输出，并下拉为低电平（不然某个上拉电阻就会对其产生影响）。当然，首先要将两块板连好，然后将其中一个单片机配置为主机以产生 SCK 信号，另一个则被配置为从机。此外还要记住，两块板子中只能有一块连接电源，另一块板则通过 PICTail 连接器取电。类似地，RB13 和 RB14 与多路复用器 U7 联用就可以通过串行接口 UART1 交换信号。

8.16 对 PIC24 行家的提示

PIC32 单片机的 SPI 接口与 PIC24 的外围设备基本相同，只是在设计中增加了一些重要的性能改进。下面列出了会对 PIC32 移植代码产生影响的一些主要差别。

（1）SPIxCON 寄存器的控制位的布局变得更加紧凑（就像其他外围设备一样）。于是，ON、FRZ 和 IDL 位现在位于标准位置（位 15、14 和 13）。它们过去都位于 SPIxSTAT 寄存器中。

（2）SPIxCON 寄存器的上半部分（现在已经扩展为 32 位）用于放置帧控制位（FRMEN、SPIFSD 等），而这些位以前位于 SPIxCON2 寄存器中。

（3）新增的 MODE32 位用于选择 32 位工作模式。

（4）SPI 模块的时钟预分频（过去是两部分，共（3+2）位）现在已扩展为完整的 9 位波特率产生模块，并由单独的寄存器 SPIxBRG 直接控制。

8.17 提示与技巧

如果要将重要数据存入外部非易失性存储器（SEE），那么就需要增加一些额外的安全措施（硬件上的和软件上的）。从硬件的角度看，要确保：

- 器件附近有充足的电源解耦（电容）；
- 片选线上设置有上拉电阻（10kΩ），以防止单片机上电和复位时出现浮动；
- 当 PIC32 的 I/O 悬空（三态）时，SCK 线上要设置额外的下拉电阻（10kΩ），以防止上电时出现外围设备时钟；
- 确保为单片机提供干净、快速的上电和掉电波形，以保证上电复位（POR）电路能够正常运行。如果有必要，还可以增加外置的电压监视器（比如 MCP809）。

采用一些软件措施可以防止一些极少出现的情况，比如程序错误或者著名的宇宙射线触发的写操作。下面是一些有关这方面的建议。

- 不要在刚刚上电时读取或更新 SEE 的内容。要等待数毫秒直到电源稳定后（等待的时间与应用系统紧密相关）再执行这类操作。
- 增加软件写使能标志，并要求应用程序在调用写操作函数前先置位该标志，从而能在随后检查一些应用程序特有的进入条件。
- 增加栈深度计数器，需要进入栈的函数在进栈时要递增该计数器，而出栈时则要递减该计数器。如果该计数器未达到预期值，就不能执行写操作函数。

- 有些用户不喜欢从头（0x0000）或者从末尾（0xffff）开始使用 SEE 存储器，他们认为这些单元显然更容易老化。
- 如果有必要，可以调用两次写操作，将每个关键的数据片保存两个副本。如果每个副本都包含校验值，或者在读回时能够进行比较，那么就很容易辨别存储器故障，并能从故障中恢复。

8.18 练习

尽管 PIC32 的 SPI 外围设备模块能使用的外围设备时钟系统的最高频率只有 50MHz，但是在 3V 供电下很少有外围设备能够运行得这么快。特别是串行 EEPROM 芯片 25LC256，在 2.5～4.5V 供电条件下允许的最快时钟频率为 5MHz。这就意味着最快的 SPI 外围设备也只需将波特率产生器配置为 1:8（36MHz/8=4.5MHz）就能满足存储器的要求。于是，连续读操作就能实现接近 4Mbit/s，或者 512KB/s 的最大数据吞吐率。即使在该速率下，PIC32 单片机仍能在每接收 1 字节数据前执行 140 条指令。这意味着在我们创建的简单的 SEE 工程中浪费了大量的处理能力，而只是在循环中坐等发送来的数据。

（1）基于中断驱动状态机开发一个更加先进的函数库，并且使用 DMA 提高 PIC32 处理能力的使用率。在第 13 章，我们将学习使用 DMA 连接 SPI 接口，尽管它不和串行 EEPROM 连接，但却是个更加通俗而有趣的应用。

（2）尝试打开 SPI 模块新的 32 位模式，以加速基本的字读/写操作。但是请注意：SEE 命令是字节宽度的，因此可能需要在 8 位和 32 位模式之间反复切换。这样做真地会节省时间/代码量吗？

8.19 参考书

F. Eady 所著的 *Networking and Internetworking with Microcontrollers*。这是一本有趣的嵌入式控制系统的串行通信入门书。作者探讨了基本的同步和异步通信接口，以实现 8 位单片机通信。

8.20 链接

www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1406&dDocName=en010003。通过该链接可以在 Microchip 公司的网站上搜索到名为 Total Endurance Software 的免费软件工具。它能帮助你估计在实际应用条件下的 NVM 器件的寿命，并给出 e/w 总次数，估计出在到达某个故障节点前的应用系统能正常工作的年数。

第 9 章　异步通信

9.1　计划

一旦去掉两个设备之间的同步串行接口的时钟信号线，它就变成了异步通信接口。无论需要全双向（全双工）通信还是半双工通信（某一时刻只能向一个方向传输）、多点通信还是点到点通信，都有很多异步协议可供选择，它们都能更有效地利用传输介质。本章将介绍 PIC32 单片机的异步串行通信模块 UART1 和 UART2，并利用它们实现基本的 RS232 接口。我们还将开发一个控制台函数库，它将在以后的工程中用于通信和调试。

9.2　准备

除了需要常用的软件工具 MPLAB IDE、MPLAB C32 编译器以及 MPLAB SIM 仿真器之外，本章还需要使用 Explorer 16 演示板、在线调试器以及一台带有 RS232 串行端口（或者是接 USB 转接器的串行端口）的 PC 机。此外，还需要终端仿真软件。如果你使用的是微软的 Windows 操作系统，那么使用超级终端应用程序（HyperTerminal，可通过菜单 Start|Programs|Accessories|Communication|Hyper Terminal 启动）就可以了。

9.3　探索

UART 接口可能是嵌入式控制领域中最古老的通信接口了。它的很多功能都是为了与第一代机械式打字机兼容而设计的，因此，其中有些技术已经有 100 多年的历史了。

另一方面，如今在新型电脑（特别是笔记本电脑）上已经很难找到异步串行端口了。串行端口已经被视为“古老的接口”，因此近几年来，电脑生产商在巨大压力的情况下已经用 USB 接口替代了串行端口。尽管串行端口已不再流行，并且 USB 接口又具有明显的性能和特性优势，但由于它极为简单并且实现成本极低，因此在嵌入式应用领域中仍在使用。

目前仍在使用的异步串行端口主要有 4 类。

（1）*RS232 点到点连接*。通常被简称为“串行端口”，广泛用于终端、调制解调器以及个人电脑，采用+12V/−12V 收发器。

（2）*RS485(EIA-485)多点串行连接*。主要用于工业领域，采用 9 比特字和特殊的半双工收发器。

（3）*LIN 总线*。这是一种用于非关键自动化应用系统的低成本、低电压总线。需要一台具有自动检测波特率功能的 UART。

（4）*红外无线通信*。需要 38~40kHz 信号调制以及特殊的光学收发器。

PIC32 单片机的 UART 单元支持上述 4 种应用，并且还有其他一些有趣功能。

为了演示 UART 外围设备的基本功能，我们将使用 Explorer 16 演示板，其中 UART2 单元与 RS232 收发器芯片相接，进而与一个 9 孔 D 型母插座相连。它能与任何一台电脑的串行端口相连，对于没有这种“古老接口”的电脑，则可以通过 RS232-USB 转换器实现连接。无论采用哪种方法，都可以使用 Windows 操作系统的超级终端程序与一台经过基本配置的 Explorer 16 演示板交换数据。

简化的 UART 模块的框图见图 9-1。

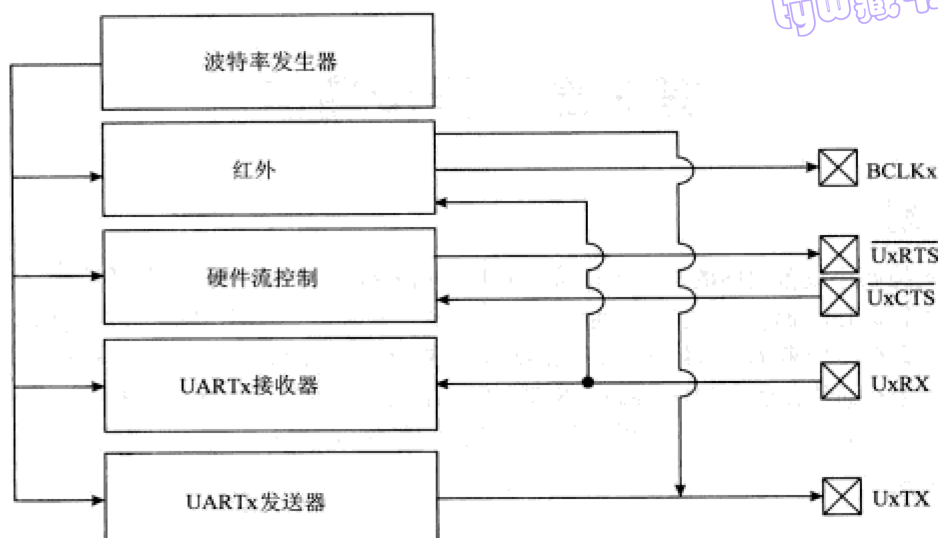


图 9-1　简化的 UART 模块的框图

第一步是配置发送器的参数，主要包括以下几个参数：

- 波特率；
- 数据位的个数；
- 校验位（如果有的话）；
- 停止位的个数；
- 握手协议。

对于我们的演示系统来说，可以选择快速而方便的配置：115200，8，N，1，CTS/RTS。也即：

- 115200 波特；
- 8 个数据位；
- 无校验；
- 1 个停止位；
- 使用 CTS 和 RTS 线作为硬件握手线。

9.4　UART 的配置

利用 New Project Setup（创建新工程）检查表创建一个名为 Serial 的新工程，并创建一个名为 serial.c 的源文件。首先添加一些有用的 I/O 定义，以控制硬件握手线：

```
/*
** Asynchronous Serial Communication
** UART2 RS232 asynchronous communication demonstration code
*/

#include <p32xxxx.h>

// I/O definitions for the Explorer16
#define CTS       _RF12               // Clear To Send, input
#define RTS       _RF13               // Request To Send, output
#define TRTS      TRISFbits.TRISF13   // Tris control for RTS pin
```

Windows 是一个多任务操作系统，它的应用程序有时可能得等待很长的一段延时，没有硬

件握手就可能导致大量数据丢失，因此与 Windows 的终端程序通信必须使用硬件握手。我们将使用一个 I/O 引脚作为输入端口（对应于 Explorer 16 板上的 RF12 引脚），以检测终端是否准备好接收新字符（Clear To Send 信号），再用一个 I/O 引脚作为输出端口（对应于 Explorer 16 板上的 RF13 脚），并在应用程序准备好接收字符时提示终端（Request To Send 信号）。

为了设置波特率，必须使用波特率发生器（U2BREG），这是一个使用外围设备总线时钟的 16 位计数器。从器件手册可以了解到，在正常工作情况下（BREGH=0），它使用的分频系数为 1:16，而在高速运行模式下（BREGH=1），它使用的分频系数为 1:4。根据数据手册上的一个简单公式，就能为本应用计算出理想的设置[参见公式（9-1）]。

公式（9-1） UxBREG=1 时 UART 的波特率

$$\text{波特率}=\frac{F_{pb}}{4\cdot(U\times BRG+1)}$$

$$U\times BRG=\frac{F_{pb}}{4\cdot\text{波特率}}-1$$

在本例中，公式（9-1）对应的表达式为：

U2BREG=(36 000 000/4/115 200)−1=77.125

由于我们最终只能使用 16 位整数，为了确定结果的准确性，需要使用反算公式来计算实际的波特率，并确定偏差的百分比：

$$\text{偏差}=\{[F_{pb}/4/(U2BREG+1)]-\text{波特率}\}/\text{波特率}\%$$

舍入值为 77 时，我们算得实际的波特率为 115 384 波特，误差仅为 0.2%，完全在允许的范围内。而当舍入值为 78 时，对应的波特率为 113 924 波特，误差增大到 1.1%，但仍然在标准 RS232 接口允许的范围内（它的允许范围是±2%）。

因此，可以定义常数 BRATE 为：

```
#define BRATE     77     // 115,200 Bd (BREGH=1)
```

此外，还可以定义两个常数，以便初始化 UART2 主控制寄存器 U2MODE 和 U2STA（参见图 9-2）。

U-0	U-0	U-0	U-0	U-0	U-0	U-0	U-0
—	—	—	—	—	—	—	—
位 31							位 24

U-0	U-0	U-0	U-0	U-0	U-0	U-0	U-0
—	—	—	—	—	—	—	—
位 23							位 16

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
ON	FRZ	SIDL	IREN	RTSMD	—	UEN<1:0>	
位 15							位 8

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
WAKE	LPBACK	ABAUD	RXINV	BRGH	PDSEL<1:0>		STSEL
位 7							位 0

图 9-2 UxMODE 控制寄存器

U2MODE 的初始值包含 BREGH 位、停止位的个数以及校验位的配置。

```
#define U_ENABLE 0x8008 // enable,BREGH=1, 1 stop, no parity
```

通过对 U2STA 初始化，可以打开发送器并清除错误标志（参见图 9-3）：

```
#define U_TX   0x0400  // enable tx, clear all flags
```

U-0	U-0	U-0	U-0	U-0	U-0	U-0	R/W-0
—	—	—	—	—	—	—	ADM_EN
位 31							位 24

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
ADDR<7:0>							
位 23							位 16

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R-0	R-1
UTXISEL<1:0>		TXINV	RXEN	TXBRK	TXEN	TXBF	TRMT
位 15							位 8

R/W-0	R/W-0	R/W-0	R-1	R-0	R-0	R/C-0	R-0
URXISEL<1:0>		ADDEN	RIDLE	PERR	FERR	OERR	RXDA
位 7							位 0

图 9-3 UxSTA 控制寄存器

下面就利用上述定义的常数来初始化 UART2 单元、波特率发生器以及用于握手的 I/O 引脚。

```
void initU2( void)
{
  U2BRG = BRATE;          // initialize the baud rate generator
  U2MODE = U_ENABLE;      // initialize the UART module
  U2STA = U_TX;           // enable the Transmitter
  TRTS = 0;               // make RTS an output pin
  RTS = 1;                // set RTS default status (not ready)
} // initU2
```

9.5 数据发送与接收

向串行端口发送字符的过程可分为 3 步。

（1）确认终端（运行 Windows 超级终端程序的 PC 机）就绪。检查 Clear to Send（CTS）线。CTS 低电平有效，因此当它为高电平时，需要等待一段时间。

（2）确认 UART 已完成之前的数据发送。PIC32 的 UART 单元有 4 级深度的 FIFO 缓存，因此需要等到至少最顶端的缓存变空为止；换句话说，需要检查发送缓冲区满标志 UTXBF 已被清零。

（3）最后，向 UART 发送缓冲区（FIFO）发送新的字符。

上述所有步骤可以方便地封装在一个函数里，称之为 putU2()，以遵从 C 语言 I/O 函数库（stdio.h）的所有字符输出函数（比如 putchar()、putc()、fputc()等）的习惯，它们都使用 put-前缀：

```
int putU2( int c)
{
  while ( CTS);                // wait for !CTS, clear to send
  while ( U2STAbits.UTXBF);    // wait while Tx buffer full
  U2TXREG = c;
  return c;
```

```
} // putU2
```

为了从串行端口接收数据，需要完成下列和发送数据类似的步骤。

(1) 通过设置 RTS 信号（低电平有效）提醒终端准备接收数据。

(2) 耐心等待数据抵达接收缓冲区，检查 UART2 状态寄存器 U2STA 里的 URXDA 标志。

(3) 从接收缓冲区（FIFO）中提取字符。

同样地，可以将上述步骤封装到一个函数里：

```
char getU2( void)
{
  RTS=0;                         // assert Request To Send !RTS
  while ( !U2STAbits.URXDA);     // wait for a new char to arrive
  RTS=1;
  return U2RXREG;                // read char from receive buffer
} // getU2
```

9.6 测试串行通信程序

为了测试串行端口控制程序，现在编写一小段程序来初始化串行端口、发送一个提示符，并且终端键盘上的输入能够回显到终端屏幕上：

```
main()
{

  char c;

  // 1. init the UART2 serial port
  initU2();

  // 2. prompt
  putU2( '>');

  // 3. main loop
  while ( 1)
  {

     // 3.1 wait for a character
     c = getU2();

     // 3.2 echo the character
     putU2( c);

  } // main loop
} // main
```

(1) 首先生成该工程，然后根据标准检查表启动调试器，并对 Explorer 16 板编程。

(2) 用串行线缆将 Explorer 16 板连接到 PC 机上（直接相连或者通过串行端口-USB 转换器连接），然后将超级终端的对应 COM 端口配置为相同的参数：115200，n，8，1，RTS/CTS。

(3) 单击超级终端的 Connect（连接）按钮，启动终端仿真器。

(4) 在 Debugger 菜单中选择 Run（运行），执行示范程序。



注解 我建议在使用 UART 时暂时不要使用 single-step（单步调试）功能，也不要使用断点或者 RunToCursor（运行到光标处）功能！具体原因请参见 9.14 节。

如果超级终端已经打开了字符回显功能，那么你将看到2个字符。为了关闭该功能，请首先单击超级终端窗口内的Disconnect按钮，然后选择菜单File|Properties打开Properties对话框，选择Settings选项卡（参见图9-4）。借此机会还可以设置更多的选项，在以后的实验中可能会用到。

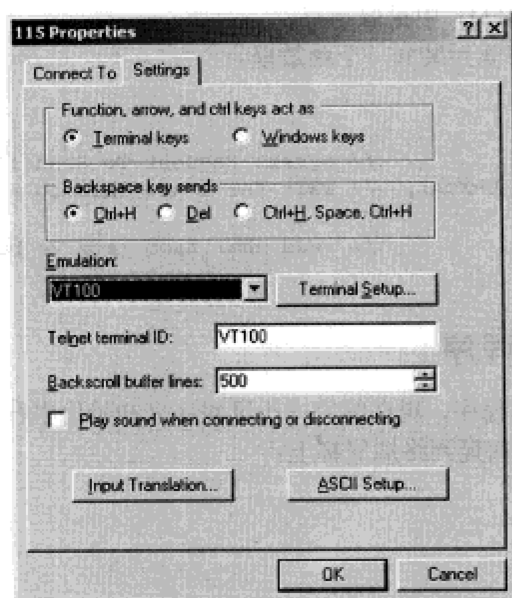


图9-4　超级终端的Properties对话框的Settings面板

（5）选择VT100 terminal（VT100终端）调试模式，以便能够使用更多的命令（可通过特殊的“转义”字符串激活），而且还能控制光标在终端屏幕上的位置。

（6）选择ASCII Setup（ASCII码设置）完成配置。特别要确保Echo typed characters locally（本地回显输入的字符）功能未被选中（这样就能立刻改善你看到的情况）。参见图9-5。

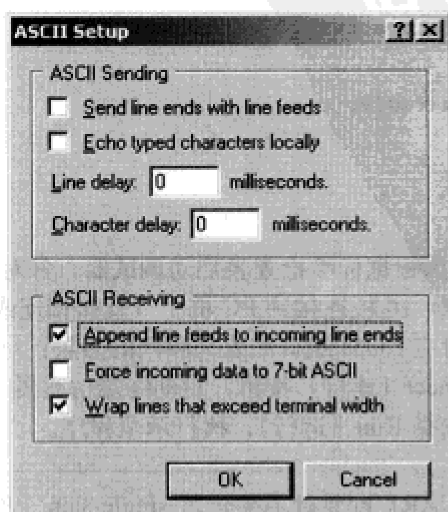


图9-5　ASCII Setup对话框

(7) 同时选中 Append line feeds to incoming line ends（将换行符附加到传入行末尾）选项。这样每次收到一个 ASCII 码的回车符（\r），超级终端就会自动附加一个换行符（\r）。

9.7 生成一个简单的控制台函数库

为了将示范工程转换成一个合适的终端控制台函数库，以便将来的工程使用，我们只需再添加两个函数就能实现：一个用于打印整个（以零为结束符）字符串，另一个用于输入整个字符行。打印字符串的函数很简单：

```
int puts( char *s)
{
  while( *s)                // loop until *s == '\0', end of string
    putU2( *s++);           // send char and point to the next one
  putU2( '\r');             // terminate with a cr / line feed
} // puts
```

它只是循环调用 putU2() 函数，从而一个接一个地将字符串中的每个字符发送至串行端口。

将来自终端（控制台）的字符串读入字符串缓冲区同样简单，但是要确保缓冲区不会溢出（用户可能会输入一个很长的字符串），并且需要将行尾的回车符转换为字符串终止符\0：

```
char *getsn( char *s, int len)
{
  char *p = s;              // copy the buffer pointer
  do{
    *s = getU2();           // wait for a new character
    if ( *s=='\r')          // end of line, end loop
      break;

    s++;                    // increment buffer pointer
    len--;
  } while ( len>1 );        // until buffer full

  *s='\0';                  // null terminate the string

  return p;                 // return buffer pointer
} // getsn
```

然而，这里所列出的函数很难投入实际使用。因为无论用户输入什么，屏幕都没有回显，因此不允许用户出错。如果有任何微小改动，都得重新输入整行。如果你像我一样，一直在进行大量的排印工作，那么键盘上磨损最严重的一定是退格键。更好版本的 getsn() 函数必须具备字符回显功能并至少准备了退格键，以便进行基本的编辑。实际上只需增加几行代码即可实现上述功能。在每次接收到字符后就立刻回显。退格字符（对应的 ASCII 码为 0x8）被解码为将缓冲区指针向后移动一个字符（直到移至行首）。此外，还需要输出一个特殊的字符串，以便从终端屏幕上清除之前输入的字符。

```
char *getsn( char *s, int len)
{
  char *p = s;              // copy the buffer pointer
  int cc = 0;               // character count
  do{

    *s = getU2();           // wait for a new character
    putU2( *s);             // echo character
```

```
    if (( *s == BACKSPACE)&&( s>p))
    {
      putU2( ' ');        // overwrite the last character
      putU2( BACKSPACE);
      len++;
      s--;                // back the pointer
      continue;
    }
    if ( *s=='\n')        // line feed, ignore it
      continue;
    if ( *s=='\r')        // end of line, end loop
      break;

    s++;                  // increment buffer pointer
    len--;
  } while ( len>1 );      // until buffer full

  *s = '\0';              // null terminate the string

  return p;               // return buffer pointer
} // getsn
```

将上述所有函数放在一个单独的文件里，并起名为 conU2.c。然后创建一个头文件 conU2.h，以决定哪些函数和常数可供外部使用或被外部可见：

```
/*
** CONU2.h
** console I/O library for Explorer16 board
*/

// I/O definitions for the Explorer16
#define CTS    _RF12     // Cleart To Send, in, HW handshake
#define RTS    _RF13     // Request To Send, out, HW handshake
#define BACKSPACE 0x8    // ASCII backspace character code

// init the serial port UART2, 115200, 8, N, 1, CTS/RTS
void initU2( void);

// send a character to the serial port
int putU2( int c);

// wait for a new character to arrive to the serial port
char getU2( void);

// send a null terminated string to the serial port
int puts( char *s);

// receive a null terminated string in a buffer of len char
char * getsn( char *s, int n);
```

9.8 测试 VT100 终端

由于已经打开了 VT100 终端仿真模式（参见前面的超级终端设置），因此现在可以使用一些命令来更好地控制终端屏幕和光标的位置，比如：

- ❑ clrscr() 函数，清除终端的屏幕；
- ❑ home() 函数，将光标移至屏幕左上角的初始位置。

这些命令可以通过发送所谓的“转义码”实现，它们都定义在 ECMA-48 标准（也称为

ISO/IEC6429 和 ANSI X3.64）中，此外还可以参考 ANSI 的退出码（exit code）。它们都以 ESC 字符（对应的 ASCII 码为 0x1b）和字符 [（左方括号）开始。

```
// useful macros for VT100 terminal emulation
#define clrscr() putsU2( "\x1b[2J")
#define home()   putsU2( "\x1b[1,1H")
```

为了测试控制台函数库，我们可以编写一小段程序，实现以下功能：

(1) 初始化串行端口；

(2) 清除终端屏幕；

(3) 发送欢迎消息/标题；

(4) 发送提示字符；

(5) 读取一整行文本；

(6) 在下一行打印读取的文本。

将下面的代码保存成一个新文件，命名为 CONU2test.c：

```
/*
** CONU2 Test
** UART2 RS232 asynchronous communication demonstration code
*/
// configuration bit settings, Fcy=72MHz, Fpb=36MHz
#pragma config POSCMOD=XT, FNOSC=PRIPLL
#pragma config FPLLIDIV=DIV_2, FPLLMUL=MUL_18, FPLLODIV=DIV_1
#pragma config FPBDIV=DIV_2, FWDTEN=OFF, CP=OFF, BWP=OFF

#include <p32xxxx.h>
#include "CONU2.h"

#define BUF_SIZE 128

main()
{
  char s[BUF_SIZE];

  // 1. init the console serial port
  initU2();

  // 2. text prompt
  clrscr();
  home();
  puts( "Exploring the PIC32!");

  // 3. main loop
  while ( 1)
  {
    // 3.1 read a full line of text
    getsn( s, sizeof(s));
    // 3.2 send a string to the serial port
    puts( s);
  } // main loop
}// main
```

(1) 利用 New Project Setup（创建新工程）检查表创建一个新工程，将 conU2.h、conU2.c 和 conU2test.c 这 3 个文件都添加到工程里，然后生成工程。

(2) 利用合适的调试器检查表来连接 Explorer 16 板，并完成编程。

（3）测试刚刚完成的新控制台函数库的编辑功能。

9.9 将串行端口用作调试工具

一旦拥有了能够通过串行端口向控制台发送和接收数据的小型函数库，也就拥有了一个新的强大的调试工具。你可以在某些位置通过调用打印函数将关键变量的内容以及其他诊断信息显示在终端上。你还能很容易地实现格式化输出，以便采用最便于阅读的格式。你还可以添加输入函数来设定参数，以便帮助你更好地测试代码，或者只是通过输入函数来暂停程序的运行，以便在需要时有时间查阅诊断输出信息。这是最古老的调试工具之一，第一台电脑就是通过与电传打印机相连实现有效调试的。

9.10 Matrix 工程

为了以一个有趣的例子结束本章，让我们一起开发一个新的示范工程吧，我们将其称为 *Matrix*。编写该程序的目的是通过向终端发送大量文本并测试性能来测试串行端口和 PC 终端仿真器的速度。唯一的问题是，我们还没有访问过大容量存储器，也没有从中提取过有意义的内容然后发送至终端。因此最后的方法是利用伪随机数发生器“产生”一些数据。stdlib.h 函数库里有一个方便的函数 `rand()`，它能返回 0 至 RAND_MAX（在 MPLAB C32 实现中，等于最大的 32 位有符号整数）之间的正整数。

利用“取模”操作符（在 C 语言中表示为%）可以将输出值减少到任意小的整数范围内，本例中则将产生对应于 ASCII 可打印字符的一个子集。例如，下面的声明就只产生 33 至 127 之间的字符：

```
putU2( 33+(rand()%94));
```

为了产生更讨人喜欢并且有趣的输出，特别是如果你恰好看过电影 *The Matrix*（《黑客帝国》），我们将逐列而不是逐行显示（随机）数据。当我们周期性地重绘整个屏幕时，将使用伪随机数发生器来改变显示内容以及每列的“长度”。

```
/*
** The UART Matrix
**
*/
#include <p32xxxx.h>
#include <stdlib.h>

#include "CONU2.h"

#define COL     40
#define ROW     23
#define DELAY   3000

main()
{
    int v[40];          // length of each column
    int i, j, k;

    // 1. initializations
    T1CON = 0x8030;  // TMR1 on, prescale 256, int clock (Tpb)
    initU2();        // initialize UART (115200, 8N1, CTS/RTS)
    clrscr();        // clear the terminal (VT100 emulation)
```

```
// 2. randomize the sequence
getU2();          // wait for a character input
srand( TMR1);     // use the current timer value as seed

// 3. init each column length
for( j = 0; j<COL; j++)
        v[j]=rand()%ROW;

// 4. main loop
while( 1)
{
  home();

  // 4.1 refresh the entire screen, one row at a time
  for( i=0; i<ROW; i++)
  {
    // 4.1.1 refresh one column at a time
    for( j=0; j<COL; j++)
    {
      // update each column

      if ( i < v[j])
        putU2( 33 + (rand()%94));
      else
        putU2( ' ');

      // additional column spacing
      putU2( ' ');
    } // for j

    // 4.1.2 empty string, advance to next line
    puts("");
  } // for i

  // 4.2 randomly increase or reduce each column length
  for( j=0; j<COL; j++)
  {
    switch ( rand()%3)
    {
      case 0: // increase length
                v[j]++;
                if (v[j]>ROW)
                 v[j]=ROW;
        break;

        case 1: // decrease length
          v[j]--;
          if (v[j]<1)
              v[j]=1;
          break;

      default:// unchanged
          break;
      } // switch
    } // for j
   } // main loop
} // main
```

别管性能，看这个程序运行就很有趣。但是显示得太快了。事实上，你得增加一小段延时（在 4.1 节的 for 循环中添加），以便看起来更舒服：

```
// 4.1.0 delay to slow down the screen update
TMR1 = 0;
while( TMR1<DELAY);
```

9.11 小结

在本章中，通过回顾 UART 单元用作 RS232 串行端口的基本功能，我们开发了一个小型控制台 I/O 函数库。我们将 Explorer 16 板与 VT100 终端（由 Windows 的超级终端模拟）相连。在后面的课程中，还将利用该函数库作为一种新的调试工具，并作为更多高级项目的用户接口。

9.12 对 C 语言编程行家的提示

我可以肯定，此时你肯定想知道能否使用 stdio.h 库定义的其他高级库函数，比如用 `printf()`控制 `stdout` 输出流传向 UART2 外围设备。我可以告诉你，不但可以，而且还能按照你想象的方式进行！

此外，stdio.h 库还定义了 2 个有用的函数：`_mon_putc()`和`_mon_getc()`，它们可用于定制标准函数库的行为。它们的声明都含有属性 `weak`，意思是 MPLAB C32 链接器允许用户重新定义它们。事实上，你可以重新定义它们来实现新的功能，比如将 SPI 端口作为输入/输出流或者重定向到 LCD 显示屏的输出等。



注解 请记住，无论你是否定制 stdio.h 里的函数，都得负责完成正确的接口初始化。因此，在首次调用 `printf()`前，请确认已打开 UART2 或者其他通信外围设备，并且设置了正确的波特率。

9.13 对 PIC 单片机行家的提示

每位嵌入式控制设计者迟早都得接触 USB 总线。如果可以暂时用一个“适配器”（它能将串行端口转换到 USB 端口）代替，那么也是可行的选择，但是最终你还得使用 USB 总线，以充分利用它的先进性能和兼容性。一些 8 位和 16 位 PIC 单片机已经集成了 USB 串行接口引擎（Serial Interface Engine，SIE），并作为标准的通信接口。Microchip 公司提供免费的 USB 软件包，其中包含适用于大多数普通应用的驱动和现成的方案。

其中之一就是通信设备类（Communication Device Class，CDC），它能使 USB 连接对 PC 应用程序完全透明，以至于连超级终端也无法分辨。最重要的是，你无需编写或安装任何特殊的 Windows 驱动。当使用 C 语言编写应用程序时，如果不是为了指定通信参数，你甚至觉察不到它们的区别。使用 USB 总线时，无需设置波特率，无需计算校验，并且通信的速度要快得多。

9.14 提示与技巧

正如在本章前面的例子中提到的，请不要在打开并使用 UART 与超级终端程序交换数据的程序中进行单步调试。你会无奈地看到超级终端程序在没有任何明显的理由的情况下出现误动作或者完全被锁死，并且忽视发给它的任何数据。

为了理解该问题，你需要了解更多有关 MPLAB ICD2 在线调试器的运行原理。当以单步调试模式每执行一条指令后或者遇到断点后，ICD2 调试器不但会停止 CPU 运行，而且会“冻结”所有的外围设备。就像突然变成极冷的冰块一样地冻结所有外围设备，数字电路里不会再

传输一个时钟脉冲。如果是正在工作的 UART 外围设备发送到中间时遇到这种情况，输出串行端口线（TX）也会被冻结到当前状态。如果这个瞬间正在移动一位数据，比如是 1，那么 TX 线就会不确定地保持为“中断”状态（低）。另一方面，超级终端程序则会发现这种永久性的“中断”，并理解为线路错误。它会认为丢失了连接，并断开当前的连接。由于超级终端是一个十分“基本”的程序，因此它不会通知用户；也不会发出鸣叫声或者给出错误信息，什么也没有；它会锁死！

如果你发现了这个潜在的问题，也没什么大不了的。用 ICD2 重启程序时，只需记住先单击超级终端的 Disconnect（断开）按钮，然后再次单击 Connect（连接）按钮即可。所有操作就都恢复正常了。

9.15 练习

（1）编写一个带有缓冲 I/O（基于中断）的控制台函数库，以便尽可能减少对程序运行（以及调试）的影响。

（2）开发一个简单的命令行解释器，它能识别一小部分定义的关键字，能通过查看和修改 RAM 存储器单元的数值或提供 Flash 存储器的十六进制数据堆来帮助调试。

9.16 参考书

J. Axelson 所著的 *Serial Port Complete*，第二版。这个新版本出版得很及时，我正好能在这里引用它。该书的作者因为 *USB Complete* 一书（下文还会提到）而闻名于世，后者被认为是适用于所有嵌入式控制程序员的参考书。长期以来，她写作了一系列专门讨论串行和并行通信接口的书。

J. Axelson 所著的 *USB Complete*，第 3 版。当你阅读我这本书的时候，新款的 PIC32MX 系列单片机很可能已经包含 USB 通信功能了。因此，我想你会喜欢这本书。Jan Axelson 的书已经出到第 3 版了。她还在继续添加内容，并仍然尽可能保持言简意赅。

F. Eady 所著的 *Implementing 802.11 with Microcontrollers: Wireless Networking for Embedded Systems Designers*。Fred 将他的幽默和经验都融入到嵌入式编程中，甚至使无线网络看起来都很简单。

9.17 链接

http://en.wikipedia.org/wiki/ANSI_escape_code。该链接提供了 VT100 超级终端仿真器实现的完整的 ANSI 转义码。

www.cs.utk.edu/~shuford/terminal/dec.html。这个网站介绍计算机历史方面的知识。这些终端我都用过。这会不会显得我很老呢？

第 10 章 LCD 显示

10.1 计划

如果你告诉我你桌上的 PC 主机旁还放着一台又大又重的 CRT 显示器，那么我一定会很惊讶。在过去的几年里，整个 PC 工业都采用了基于新技术的平板 LCD，它的面板尺寸更大、分辨率也更高。同时，在嵌入式控制领域也发生了类似的变化。7 段 LED 显示器已经是 20 世纪 90 年代的古董了！小型 LCD 显示屏已被普遍使用，它比 LED 的功耗更低，而且支持字符式输出（即显示文字），甚至还支持图形显示。但是，现在可能已经出现新一代的有机 LED 显示器（OLED）了，它们正准备重新夺回市场。

本章将学习如何与低成本的小型 LCD 字符式显示模块通信。借此机会，我们还可以学习使用 PMP（Parallel Master Port，并行主端口），所有 PIC32MX 单片机都提供该灵活的并行接口。

10.2 准备

除了需要常见的软件工具 MPLAB IDE、MPLAB C32 编译器以及 MPLAB SIM 仿真器之外，本章还需要使用 Explorer 16 演示板以及你所选的在线调试器（PIC32 Starter Kit、ICD2、REAL ICE 等）。

10.3 探索

Explorer 16 板可以使用 3 种不同类型的点阵、字符式 LCD 显示模块以及一种图形式 LCD 显示模块。它默认采用简单的“2 行乘 16 字符”显示方式和 3V 供电的字符式 LCD 模块相接，这种 LCD 模块和工业标准的 HD44780 控制器兼容（最常用的是 Tianma 公司的 TM162JCAWG1）。这些 LCD 模块是由 LCD 灯、行列复用驱动器、电源电路以及智能控制器组成的显示系统，所有电路都通过玻璃覆晶（Chip On Glass，COP）封装工艺组装在一起。幸亏集成度很高，控制点阵显示的电路才非常简单。我们根本不必使用数百个引脚通过行列驱动器与每个像素直接相连，而只需 11 个 I/O 端口，通过简单的 8 位并行总线与显示模块相连即可。

特别是字符式 LCD 模块（参见图 10-1），我们可以直接把 ASCII 字符码放入 LCD 模块控制器的 RAM 缓存中（即所谓的显示器数据 RAM 缓存，简记为 DDRAM）。然后，集成的字符生成器（其实是一个字符表）用 5×7 的像素点阵表示一个字符，并产生输出图像。该字符表（参见图 10-2）通常包括扩展的 ASCII 字符集，也就是说，它还包含了一些日语片假名字符以及常用的符号。尽管这个字符生成表通常存储在显示控制器 ROM 中，但是，各种显示模型都提供了不同的字符集扩展方法，可以通过访问另一个内部小 RAM 缓存（字符生成器 RAM 缓存，简记为 CGRAM）来修改或创建一些（2~8 个）新字符。

10.4 与 HD44780 控制器兼容

前面已经提到，Explorer 16 演示板使用的 2×16 LCD 模块是市场上最常用的 LCD 显示模块，它能配置成 1~4 行，每行 8、16、20、32 甚至 40 个字符，并且和如今被视为工业标准的 HD44780 芯片组兼容。

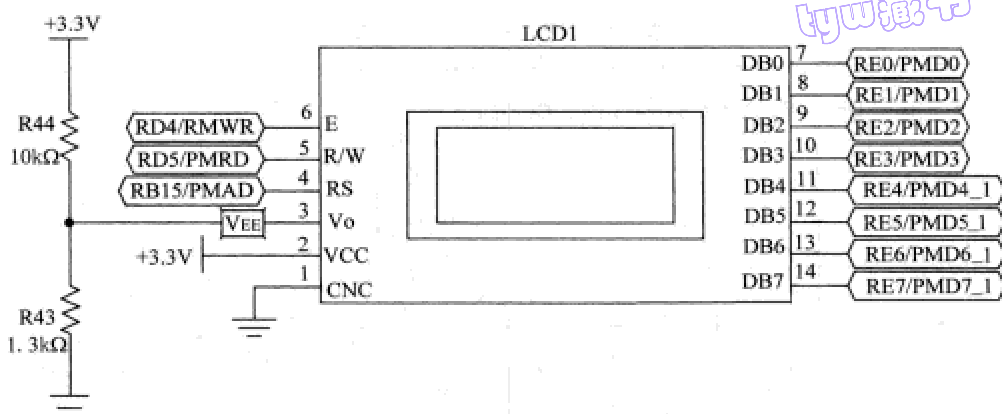


图 10-1 默认的字符式 LCD 模块的连接图

字符码	0000	0010	0011	0100	0101	0110	0111	1010	1011	1100	1101	1110	1111
××××0000			0	@	P	`	p		ー	タ	ミ	α	p
××××0001		!	1	A	Q	a	q	。	ア	チ	ム	ä	q
××××0010		"	2	B	R	b	r	「	イ	ツ	メ	β	θ
××××0011		#	3	C	S	c	s	」	ウ	テ	モ	ε	∞
××××0100		$	4	D	T	d	t	、	エ	ト	ヤ	μ	Ω
××××0101		%	5	E	U	e	u	・	オ	ナ	ユ	σ	ü
××××0110		&	6	F	V	f	v	ヲ	カ	ニ	ヨ	ρ	Σ
××××0111		'	7	G	W	g	w	ァ	キ	ヌ	ラ	g	π
××××1000		(	8	H	X	h	x	ィ	ク	ネ	リ	√	x̄
××××1001		)	9	I	Y	i	y	ゥ	ケ	ノ	ル	⁻¹	y
××××1010		*	:	J	Z	j	z	ェ	コ	ハ	レ	j	千
××××1011		+	;	K	[	k	{	ォ	サ	ヒ	ロ	×	万
××××1100		,	<	L	¥	l	|	ャ	シ	フ	ワ	¢	円
××××1101		-	=	M	]	m	}	ュ	ス	ヘ	ン	£	÷
××××1110		.	>	N	^	n	→	ョ	セ	ホ	゛	ñ	
××××1111		/	?	O	_	o	←	ッ	ソ	マ	゜	ö	█

图 10-2 与 HD44780 兼容的 LCD 显示控制器使用的字符生成器表

与 HD44780 兼容就意味着该集成控制器包括两个独立的可寻址的 8 位寄存器：一个用于传输 ASCII 数据，另一个用于传输命令和状态信息。表 10-1 和表 10-2 给出了用于配置和控制显示方式的标准命令集。

具备这种通用性的好处是，为了驱动 Explorer 16 演示板上的 LCD 而开发的代码，也可用于控制任何其他与 HD44780 兼容的字符式 LCD 显示模块。

表 10-1 HD44780 的指令集

指令	代码										描述	执行时间
	RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0		
清屏	0	0	0	0	0	0	0	0	0	1	清除显示屏，并将光标返回初始位置（0号地址）	1.64ms
光标回到初始位置	0	0	0	0	0	0	0	0	1	*	将光标返回初始位置（0 号地址）。把移位显示的内容返回到初始位置，DDRAM的内容保持不变	1.64ms
进入模式设置	0	0	0	0	0	0	0	1	I/D	S	设定光标的移动方向（I/D），指定移动显示（S）。这些操作可以在读/写数据时进行	40μs
显示打开/关闭控制	0	0	0	0	0	0	1	D	C	B	设置打开/关闭所有的显示（D）、打开/关闭光标（C）以及打开/关闭光标位置符的闪烁	
光标/显示移动	0	0	0	0	0	1	S/C	R/L	*	*	设置光标移动或者显示移动（S/C），移动方向（R/L）。DDRAM 的内容保持不变	40μs
功能设置	0	0	0	0	1	DL	N	F	*	*	设置接口数据宽度（DL）、显示的行数（N）以及字符的字体（F）	
设置 CGRAM 地址	0	0	0	1	CGRAM 的地址						设置 CGRAM 的地址，完成该设置后才能发送和接收 CGRAM 数据	40μs
设置 DDRAM 地址	0	0	1	DDRAM 的地址							设置 DDRAM 的地址，完成该设置后才能发送和接收 DDRAM 数据	40μs
读取忙标志和地址计数器	0	1	BF	CGRAM 或 DDRAM 的地址							读取忙标志（BF），它指示正在进行内部操作，还能读取 CGRAM 或 DDRAM 地址计数器的值（与前一条指令有关）	0μs
写 CGRAM 或 DDRAM	1	0	写数据								向 CGRAM 或者 DDRAM 写数据	40μs
读 CGRAM 或 DDRAM	1	1	读数据								从 CGRAM 或者 DDRAM 读数据	40μs

表 10-2 HD44780 的命令位

位名称	设置/状态	
I/D	0=递减光标的位置	1=递增光标的位置
S	0=禁止移动显示	1=允许移动显示
D	0=关闭显示	1=打开显示
C	0=关闭光标	1=打开光标
B	0=关闭光标闪烁	1=打开光标闪烁
S/C	0=移动光标	1=移动显示
R/L	0=向左移动	1=向右移动
DL	0=4 位接口	1=8 位接口
N	0=1/8 或者 1/11 的占空比（1 行）	1=1/16 的占空比（2 行）
F	0=5×7 点	1=5×10 点
BF	0=可以接受指令	1=正在进行内部操作

10.5 并行主端口

显然，这些显示模块共用的 8 位总线十分简单。除了 8 根双向数据线（通过启动特殊的“半字节”方式，还能再节省 4 根 I/O 线），还包括：

- ❑ 使能选通线（E）；
- ❑ 读/写选择线（R/W）；
- ❑ 用于选择寄存器的地址线（RS）。

尽管可以通过人工访问 PORTE 和 PORTD 的各个引脚来控制这 11 个 I/O 信号以实现每种总线时序，但是我们并不打算采用这种方式，而是利用 PIC24 架构引入的并在 PIC32 架构中继续增强的新外围设备：并行主接口（Parallel Master Port，PMP）。这种可寻址的并行端口能方便地访问大量常用的外部并行设备，包括模-数转换器、RAM 缓冲、与 ISA 总线兼容的接口、LCD 显示模块，甚至还有硬盘驱动器和 CF 卡。

你可以将 PMP 看作 PIC32 架构中增添的一种灵活的 I/O 总线，它能把单片机从控制慢速外围设备的繁琐任务中解脱出来。PMP 包括：

- ❑ 8 位或者 16 位双向数据通道；
- ❑ 多达 64KB 的寻址空间（16 根地址线）；
- ❑ 6 根选通/控制线，包括 1 根使能线、1 根地址锁存线、读和写线（可以是单独的两根线，也可以复用同一根线）和 2 根片选线。

PMP 还能配置成工作在从模式下，并作为一种可寻址的外围设备附加到更大的微处理器/微控制器系统中。

为了使所选的控制信号和极性与目标总线相匹配，并且能够微调时序以适应与之相连的外围设备的速度，总线读与总线写时序都是完全可编程的。

10.6 配置 PMP 用于 LCD 模块控制

与其他 PIC32 外围设备一样，配置 PMP 也需要使用一组专用的控制寄存器。第一个也是最重要的寄存器是 PMCON，它的控制位顺序与其他的 xxCON 寄存器类似（参见图 10-3）。

U-0	U-0	U-0	U-0	U-0	U-0	U-0	U-0
—	—	—	—	—	—	—	—
位 31							位 24

U-0	U-0	U-0	U-0	U-0	U-0	U-0	U-0
—	—	—	—	—	—	—	—
位 23							位 16

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	U-0	R/W-0	R/W-0
ON	FRN	SIDL	ADRMUX1	ADRMUX0	—	PTWREN	PTRDEN
位 15							位 8

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	U-0	R/W-0	R/W-0
CSF1	CSF0	ALP	CS2P	CS1P	—	WRSP	RDSP
位 7							位 0

图 10-3 PMCON 控制寄存器

这次需要初始化的控制寄存器比较多，包括 PMMODE、PMADDR、PMSTAT 以及 PMAEN。

它们都包含强大的功能选项，因此需要你仔细研究如何使用。这里不打算逐一地介绍它们，而是仅列出与 LCD 模块相连时需要使用的关键选项：

- ❑ 启用 PMP 单元；
- ❑ 全双工接口（需要使用独立的数据和地址线）；
- ❑ 使能选通信号（在 RD4 引脚上）；
- ❑ 读信号（在 RD5 引脚上）；
- ❑ 使能选通信号高电平有效；
- ❑ 读信号高电平有效，写信号低电平有效；
- ❑ 主模式，读写信号共用同一个引脚（RD5）；
- ❑ 8 位总线接口（使用 PORTE 引脚）；
- ❑ 只需一个地址位，因此选最小配置，包括 PMA0（在 RB15 引脚上）和 PWA1（不用）。

此外，考虑到典型的 LCD 单元的传输速度都非常慢，因此最好选择最为普通的时序，并增加在读写时序中每个阶段允许的最大等待状态个数：

- ❑ 在读/写前有 $4 \times T_{pb}$ 等待数据建立；
- ❑ 在 R/W 和使能之间等待 $15 \times T_{pb}$；
- ❑ 在使能后有 $4 \times T_{pb}$ 等待数据建立。

10.7 访问 LCD 显示模块的小型函数库

利用 New Project Setup（创建新工程）检查表建立一个名为 Liquid 的工程以及一个名为 liquid.c 的源文件，然后开始创建小型 LCD 接口函数库。

首先编写 LCD 初始化程序。很自然地要先初始化 PMP 端口的关键控制寄存器：

```
void LCDinit(void)
{

  // PMP initialization
  PMCON = 0x83BF;            // Enable the PMP, long waits
  PMMODE = 0x3FF;            // Master Mode 1
  PMPEN = 0x0001;            // PMA0 enabled
```

接着就能与 LCD 模块进行首次通信了，我们可以按照 LCD 生产商推荐的标准的 LCD 初始化流程进行初始化操作。初始化流程具有严格的时间要求（具体参见 HD44780 的指令集），并且必须在 LCD 模块进行内部初始化（上电复位）的 30ms 时间之后进行。为了简单和可靠起见，可以复制一段延时程序到 LCD 模块初始化函数中，并在所有的初始化过程中用定时器 1 单元实现简单而精确的定时循环：

```
// init TMR1
  T1CON = 0x8030;            // Enabled,1:256 Fpb, 1 tick ~ 6us

  // wait for >30ms
  TMR1 = 0; while(TMR1<6000); // 6000 x 6us = 36ms
```

为了方便使用，定义了一组常数以提高接下来的代码的可读性：

```
#define LCDDATA 1          // RS = 1 ; access data register
#define LCDCMD 0           // RS = 0 ; access command register
#define PMDATA PMDIN1      // PMP data buffer
```

向 LCD 模块发送指令时，首先要选中命令寄存器（将地址线设为 PMA=RS=0）（参见图 10-4）。

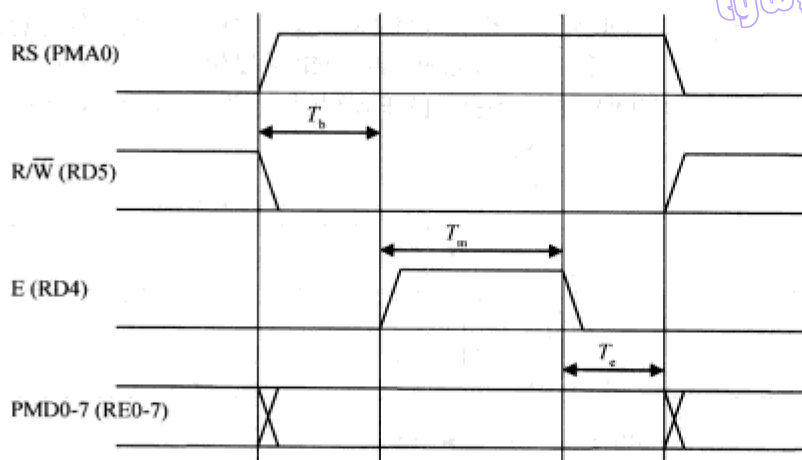


图 10-4　PMP 与 LCD 显示模块的 8 位接口的写命令时序

然后将命令字放入 PMP 数据输出缓冲器，启动 PMP 写操作：

```
PMADDR = LCDCMD;            // command register (ADDR = 0)
PMDATA = 0x38;              // set: 8-bit interface, 2 lines, 5x7
```

PMP 将依照如下顺序实现完整的总线写操作。

(1) 将地址放在 PMP 地址总线上（PMA0）。

(2) 将 PMDATA 的内容放在 PMP 数据总线上（PMD0~PMD7）。

(3) R/W 信号置低（RD5）。

(4) 等待 $4\times T_{pb}$（T_b）后选通信号 E 置高。

(5) 再等待 $15\times T_{pb}$（T_m）后使能选通线置低。

(6) 再等待 $4\times T_{pb}$（T_e）后释放数据总线。

请注意这个操作过程的时间很长，从 PIC32 开始启动到结束共需花费 $10\times T_{pb}$ 的时间（相当于超过 0.5ms）。换句话说，当 PIC32 已经又执行了超过 40 条指令时，PMP 还在忙于处理当前的操作。由于 LCD 模块执行命令需要的实际时间比这还长得多（大于 40μs），因此不必担心 PMP 完成一条命令所花费的时间太长，只要耐心等待就可以了。

```
TMR1 = 0; while( TMR1<8);    // 8 x 6us = 48us
```

下面将采用类似的方法完成 LCD 模块初始化过程中的其他操作：

```
PMDATA = 0x0c;                // ON, no cursor, no blink
TMR1 = 0; while( TMR1<8);     // 8 x 6us = 48us
PMDATA = 0x01;                // clear display
TMR1 = 0; while( TMR1<300);   // 300 x 6us = 1.8ms

PMDATA = 0x06;                // increment cursor, no shift
TMR1 = 0; while( TMR1<300);   // 300 x 6us = 1.8ms
```

完成 LCD 模块初始化后，事情就变得简单多了，可以使用 LCD 模块的读取忙标志（Read Busy Flag）命令而不必使用定时循环。该标志位能够指示出集成 LCD 模块控制器是否已经完成上一条命令并准备接收和处理新命令了。为了读取包含 LCD 忙标志位的 LCD 状态寄存器，PMP 需要执行一次总线读操作。这个过程分为两步：先通过读取（并丢弃）PMP 数据缓存

（PMPDIN）中的内容来启动读取时序；然后，当 PMP 操作完成时，数据缓存包含了来自总线的实际值，再次读取 PMP 数据缓存。但是如何判断 PMP 读操作已经完成了呢？

很简单，可以检查 PMMODE 控制寄存器的 PMP 忙标志位（PMMODEbits.BUSY），参见图 10-5。

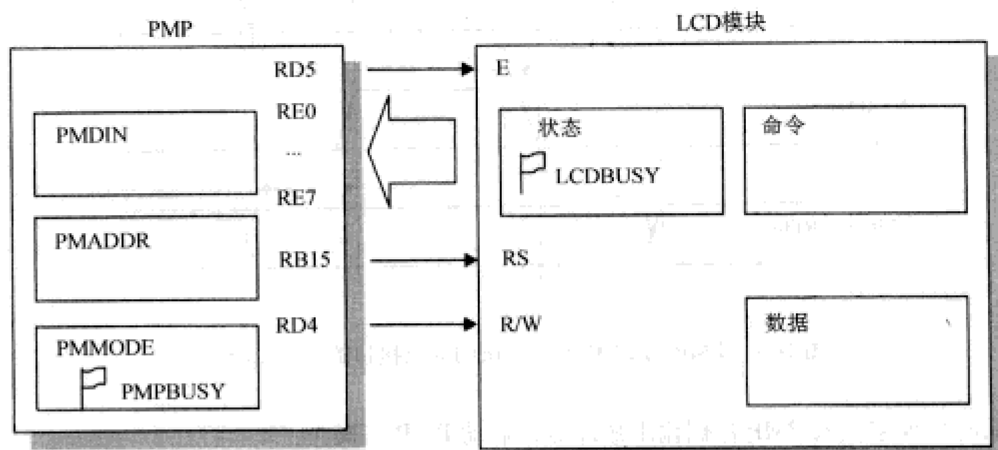


图 10-5　PMP 与 LCD 模块的连接原理图

总之，为了检查 LCD 模块的忙标志位，首先要检查 PMP 忙标志，以确保之前发送的命令已完成，然后发送一条读命令，再次等待 PMP 忙标志，而正是这时候才能访问实际 LCD 模块的状态寄存器的内容，其中就包括 LCD 忙标志。

把寄存器地址以参数形式传递给读取函数，可以让读取函数更加通用，从而能读取 LCD 的状态寄存器或者数据寄存器，相关代码如下：

```
char readLCD( int addr)
{
    int dummy;
    while( PMMODEbits.BUSY);    // wait for PMP to be available
    PMADDR = addr;              // select the command address
    dummy = PMDATA;             // init read cycle, dummy read
    while( PMMODEbits.BUSY);    // wait for PMP to be available
    return( PMDATA);            // read the status register

} // readLCD
```

LCD 模块的状态寄存器包含两部分信息：LCD 忙标志以及 LCD RAM 指针的当前值。我们可以通过两个简单的宏定义 busyLCD() 和 addrLCD() 来分离这两部分信息，再用第三个宏定义 getLCD() 访问数据寄存器：

```
#define busyLCD() readLCD( LCDCMD) & 0x80
#define addrLCD() readLCD( LCDCMD) & 0x7F
#define getLCD() readLCD( LCDDATA)
```

利用 busyLCD() 可以创建一个向 LCD 模块写数据或者命令的函数：

```
void writeLCD( int addr, char c)
{
    while( busyLCD());
```

```
    while( PMMODEbits.BUSY);    // wait for PMP to be available
    PMADDR = addr;
    PMDATA = c;
} // writeLCD
```

再定义一些宏就构成了完整的函数库。

- putLCD()，用于向 LCD 模块发送 ASCII 数据：

```
#define putLCD( d)  LCDwrite( LCDDATA, (d))
```

- cmdLCD()，用于向 LCD 模块发送通用命令：

```
#define cmdLCD( c)  writeLCD( LCDCMD, (c))
```

- homeLCD()，可将光标复位到第一行第一个字符处：

```
#define homeLCD()   writeLCD( LCDCMD, 2)
```

- clrLCD()，可清除全部显示内容：

```
#define clrLCD()    writeLCD( LCDCMD, 1)
```

最后，为了方便使用，我还想添加一个 putsLCD()函数，它能将整个非空的字符串发送到显示模块：

```
void putsLCD( char *s)
{
    while( *s)
        putLCD( *s++);
}//putsLCD
```

下面就将我们设计的函数组合起来，再加一个很短的主函数：

```
main( void)
{
    // initializations
    initLCD();

    // put a title on the first line
    putsLCD( "Exploring       ");

    // put the cursor on the second line (addr 0x40)
    cmdLCD( 0x80 | 0x40);
    putsLCD( "        the PIC32");

    // main loop, empty for now
    while ( 1)
    {
    }
} // main
```

如果一切正常，那么在生成工程并用所选的调试器烧录 Explorer 16 演示板之后，就能看到 LCD 显示屏上分两行显示了标题字符串。

10.8 生成 LCD 函数库并使用 PMP 函数库

只要包含 pmp.h 库或者只是包含 plib.h 头文件，就能用特定的 PMP 外围设备函数库来实现与前面相同的功能。只需 4 个函数就能控制 PMP 并实现与 LCD 显示模块的对话，它们是：

- mPMPOpen()，用于配置并行主端口；
- PMPSetAddress()，用于设定地址寄存器；
- PMPMasterWrite()，用于初始化基本的写操作；
- mPMPMasterReadByte()，可以初始化基本的读操作，并返回一字节的数据。

知道了这些函数后，我们不但要重写代码以使用更具描述性的宏定义以及函数库提供的定义，而且还要重新整理代码，将它转换成一个实用的小型函数库；不久以后这个函数库就将用在 Explorer 16 演示板上设计的其他工程上。

首先要新建一个名为 LCD library 的工程，并新建一个名为 LCDlib.c 的源文件。下面是用 PMP 库函数和宏定义实现的 initLCD() 函数：

```
void initLCD( void)
{
    // PMP initialization
    mPMPOpen( PMP_ON | PMP_READ_WRITE_EN | 3,

              PMP_DATA_BUS_8 | PMP_MODE_MASTER1 |
              PMP_WAIT_BEG_4 | PMP_WAIT_MID_15 |
              PMP_WAIT_END_4,
              0x0001,          // only PMA0 enabled
              PMP_INT_OFF);    // no interrupts used

    // wait for >30ms
    Delayms( 30);

   //initiate the HD44780 display 8-bit init sequence
    PMPSetAddress( LCDCMD);       // select command register
    PMPMasterWrite( 0x38);        // 8-bit int, 2 lines, 5x7
    Delayms( 1);                  //>48us

    PMPMasterWrite( 0x0c);        // ON, no cursor, no blink
    Delayms( 1);                  //>48us

    PMPMasterWrite( 0x01);        // clear display
    Delayms( 2);                  //>1.6ms

    PMPMasterWrite( 0x06);        // increment cursor, no shift
    Delayms( 2);                  //>1.6ms
} // initLCD
```

请注意，为了使用一个简单的以 1 毫秒为基本递增单位的延时函数 Delayms()，在初始化部分故意夸大了延时时间。我们很快就会看到该延时函数是在哪里定义以及如何定义的。

下面是我们设计的简单 LCD 函数库中包含的其他核心函数：

```
char readLCD( int addr)
{
    PMPSetAddress( addr);         // select register
    mPMPMasterReadByte();         // initiate read sequence
    return mPMPMasterReadByte();  // read actual data
} // readLCD

void writeLCD( int addr, char c)
{
    while( busyLCD());
    PMPSetAddress( addr);         // select register
    PMPMasterWrite( c);           // initiate write sequence
} // writeLCD
```

如果你觉得前面的工程（Liquid）中将光标停放在显示屏的第二行有些难看，那么可以对 putsLCD() 函数做一些修改。特别是让子程序理解一些特殊字符，比如行结束符（line end）、制表符（tab）以及换行符（new line）等，就像串行端口或控制台一样。

```
void putsLCD( char *s)
{
    char c;
while( *s)
{
    switch (*s)
    {
    case '\n':              // point to second line
      setLCDC( 0x40);
      break;
    case '\r':              // home, point to first line
      setLCDC( 0);
      break;
    case '\t':              // advance next tab (8) positions
      c = addrLCD();
      while( c & 7)
      {
          putLCD( ' ');
          c++;
      }
      if ( c > 15)          // if necessary move to second line
          setLCDC( 0x40);
      break;
    default:                // print character
      putLCD( *s);
      break;
    } //switch
    s++;
  } //while
} //putsLCD
```

这样，当显示一个包含（或者结尾是）字符\n（换行符）的字符串时，就会将光标放在LCD显示屏的第二行的起始位置。字符\r（行结束符）则会使光标退回第一行的起始位置，而字符\t（制表符）也能产生预期的效果。

再加上一些标准的头文件以及一些#include 语句就完成了这个源文件：

```
/*
** LCDlib.c
*/
#include <p32xxxx.h>
#include <plib.h>
#include <explore.h>
#include <LCD.h>

#define PMDATA PMDIN
```

保存刚刚完成的 LCDlib.c 文件，然后在 MPLAB 编辑窗口中新建一个源文件——头文件LCD.h，在其中声明所有需要的宏以及函数原型，这样就完成了函数库设计：

```
/*
**
** LCD.h
**
*/
#define HLCD    16      // LCD width=16 characters
#define VLCD    2       // LCD height=2 rows
```

```
#define LCDDATA 1        // address of data register
#define LCDCMD 0         // address of command register

void initLCD( void);
void writeLCD( int addr, char c);
char readLCD( int addr);

#define putLCD( d)  writeLCD( LCDDATA, (d))
#define cmdLCD( c)  writeLCD( LCDCMD, (c))

#define clrLCD()    writeLCD( LCDCMD, 1)
#define homeLCD()   writeLCD( LCDCMD, 2)

#define setLCDG( a) writeLCD( LCDCMD, (a & 0x3F) | 0x40)
#define setLCDC( a) writeLCD( LCDCMD, (a & 0x7F) | 0x80)

#define busyLCD()   ( readLCD( LCDCMD) & 0x80)
#define addrLCD()   ( readLCD( LCDCMD) & 0x7F)
#define getLCD()    readLCD( LCDDATA)

void putsLCD( char *s);
```

最后，为了测试新建的 LCD 函数库，我们要编写一小段新的测试程序——LCDlib test.c：

```
/*
** LCDlib test
**
*/
// configuration bit settings, Fcy=72MHz, Fpb=36MHz
#pragma config POSCMOD=XT, FNOSC=PRIPLL
#pragma config FPLLIDIV=DIV_2, FPLLMUL=MUL_18, FPLLODIV=DIV_1
#pragma config FPBDIV=DIV_2, FWDTEN=OFF, CP=OFF, BWP=OFF

#include <p32xxxx.h>

#include <LCD.h>

main()
{
    initLCD();

    clrLCD();
    putsLCD( "Exploring \nthe \tPIC32");

    while( 1);
}
```

10.9 函数库 EXPLORER.C

为了将 PIC32 初始化到最佳性能(参见第 7 章)、使用向量中断(参见第 5 章)以及使用 Explorer 16 演示板提供的功能（比如 LED 显示灯，参见第 1～3 章），首先要将手头的一些函数合成一个小型函数库。在后续几章中我们将逐步添加新函数到该函数库中，这里是它的第一个版本：

```
/*
** Explore.c
**
*/

#include <p32xxxx.h>
#include <plib.h>
#include <explore.h>
```

```
void initEX16( void)
{
  // 1. disable the JTAG port to make the LED bar
  // available if not using the Starter Kit
#ifndef PIC32_STARTER_KIT
  mJTAGPortEnable( 0);
#endif

  // 2. Sysytem config performance
  SYSTEMConfigPerformance( FCY);

  // 7. allow vectored interrupts
  INTEnableSystemMultiVectoredInt();    // Interrupt vectoring

  // 8. PORTA output LEDs0..6, make RA7 an input button
  LATA = 0;
  TRISA = 0xFF80;

} // initEX16

//
void _general_exception_handler( unsigned c, unsigned s)
{
  while (1);
} // exception handler
//

/*
** Simple Delay functions
**
** uses:    Timer1
** Notes:   Blocking function
*/

void Delayms( unsigned t)
{
  T1CON = 0x8000;      // enable TMR1, Tpb, 1:1
  while (t--)
  {  // t x 1ms loop
        TMR1 = 0;
        while (TMR1 < FPB/1000);
    }
} // Delayms
```

对应的头文件 explore.h 还包含一些重要的定义和两个函数原型：

```
/*
** Explore.h
**
*/
#define FALSE    0
#define TRUE     !FALSE
#define FCY      72000000L
#define FPB      36000000L

// uncomment the following line if using the PIC32 Starter Kit
//#define PIC32_STARTER_KIT

// function prototypes
void initEX16( void);
void Delayms( unsigned);
```

10.10　创建 include 和 lib 目录

为了有序地保存已创建的文件，并使工程看起来整洁，这里要制订一些规定，将目前已创建的简单函数库分成两组，分别保存在两个子目录中。

- *include* 目录。这里存放所有.h 文件；这些文件用于声明需要使用的简单函数，包括 explore.h、LCD.h、conU2.h 和 SEE.h。
- *lib* 目录。这里存放对应的.c 文件，包括 explore.c、LCD.c、conU2.c 和 SEE.c。

从现在开始，只要将 *include* 目录添加到每个新工程的 *include search path*（头文件搜索目录）中，就能自动地引用这些头文件。具体实施步骤如下。

（1）选择菜单 Project | BuildOptions… | Project，打开 Build Option 对话框（参见图 10-6）。

（2）在 Show Directories for 下拉菜单中选择 Include Search Path。

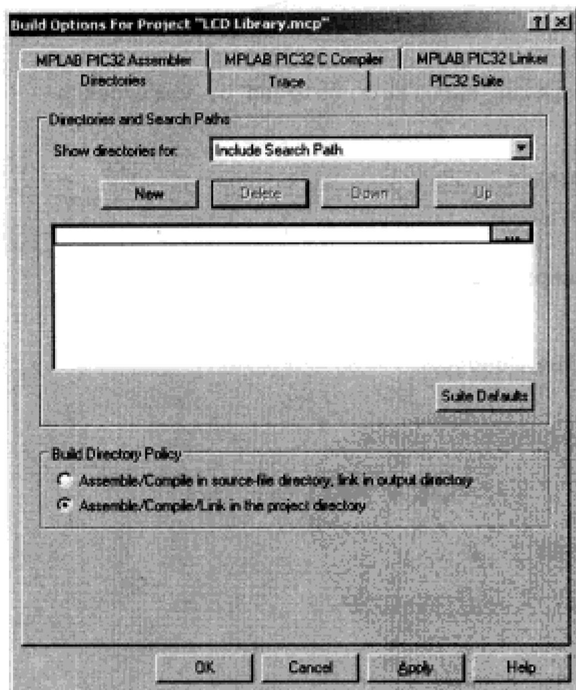


图 10-6　工程的 Build Option 对话框

（3）单击 New 按钮，新建一个空条目。

（4）单击最右边的…按钮打开 Browse For Folder 对话框（参见图 10-7）。

（5）选择新建的 *include* 目录。

（6）单击 OK 按钮关闭对话框。

（7）单击 OK 按钮接受新的设置。

（8）选择 Project|SaveProject 选项保存工程。

通过这些设置，就能以默认的引用语句引用 LCD.h 文件，而不必添加该文件实际存储位置的详细路径地址，即：

```
#include <LCD.h>
```



图 10-7　Browse For Folder 对话框

> **注解**　请注意尖括号（<>）和双引号（“”）的用法。它们的区别是编译器搜索引用文件的位置不同。我们在前面所有工程中都采用双引号法，它要求编译器在当前工程目录中搜索。而尖括号法则要求编译器在 *include search path* 中定义的一系列地址中搜索，这些地址通常包含编译器特定的（MPLAB C32）所有函数库目录（在安装软件时定义），此外还包括 Include Search Path 对话框中添加的目录。

10.11　高级 LCD 控制

如果你感觉前面的讨论太简单，还不过瘾，那么下面就要谈一些更加有趣、难度更大的技术。

在介绍 HD44780 兼容型字符式 LCD 模块时，曾提到过 LCD 模块控制器是如果利用 ROM 中的字符表和字符生成器产生显示内容的。此外，还提到过可以利用额外的 RAM 缓存（即 CGRAM）产生用户自定义字符以构成扩展字符集。根据 LCD 显示模块型号的不同，CGRAM 可以产生 2 到 8 种新字符。当然，如果有 32 个用户自定义字符，那么就几乎可以将整个字符式显示屏转变成图形式显示屏。遗憾的是，最流行并且平价的 LCD 模块，特别是 Explorer 16 演示板上使用的这一块，只支持 2 个用户自定义字符。不过我们仍旧可以用它们完成很多有趣的事情。比如接下来，我们将使用其中的一个自定义字符来演示开发一个简单的进度条。

我们需要一个函数，通过 Set CGRAM Address 命令，把 LCD 模块的 RAM 缓存指针指向 CGRAM 区域的起始位置，或者最好定义一个使用 `writeLCD()` 函数的宏：

```
#define setLCDG( a) writeLCD( LCDCMD, (a & 0x3F) | 0x40)
```

一旦缓存指针指向 CGRAM，特别是指向缓冲区的起始处（setLCDG(0)），就可以使用 putLCD()函数向缓存中写入 8 字节数据。每字节中有 5 位（低 5 位）用于表示由 8 行点阵组成的新字符。只要将缓存指针重置到 DDRAM 区域（利用宏 `setLCDC(0)`），就能使用刚才定义的字符，它的 ASCII 码为 `0x00`。

请注意，按照惯例，尽管显示屏的第一行对应的 DDRAM 缓存中地址为 0~15，但是第二

行对应的地址则始终是 0x40~0x4f，这与显示屏的宽度（实际的显示屏每行能显示的字符个数）无关。

10.12　进度条工程

下面将开始今天的最后一个工程，我们称之为 Progress（进度）。请使用 New Project Setup（创建新工程）检查表创建该工程，并且最后在 *include search path* 栏中添加 *include* 目录。

通过插入标准的模板和 *include* 声明，就能立刻创建新的源文件 ProgressBar.c。

```
/*
**
** Progress Bar
**
*/
// configuration bit settings, Fcy = 72MHz, Fpb = 36MHz
#pragma config POSCMOD = XT, FNOSC = PRIPLL
#pragma config FPLLIDIV = DIV_2, FPLLMUL = MUL_18, FPLLODIV = DIV_1
#pragma config FPBDIV = DIV_2, FWDTEN = OFF, CP = OFF, BWP = OFF
#include <p32xxxx.h>
#include <explore.h>
#include <LCD.h>
```

我们可以利用由 16 个“砖块”字符组成的字符串绘制一个块状进度条。该砖块字符可以从 LCD 字体表中选择代码 0xff 获得，它将显示一个 5×8 的黑色点阵。但是，为了实现更高的分辨率和更光滑的动态效果，我们可以利用刚刚学会的用户自定义字符功能开发一个新砖块。用（5×8）砖块构成进度条的大部分，再定义一个具有一定厚度的新字符作为进度条的末梢（参见图 10-8）。

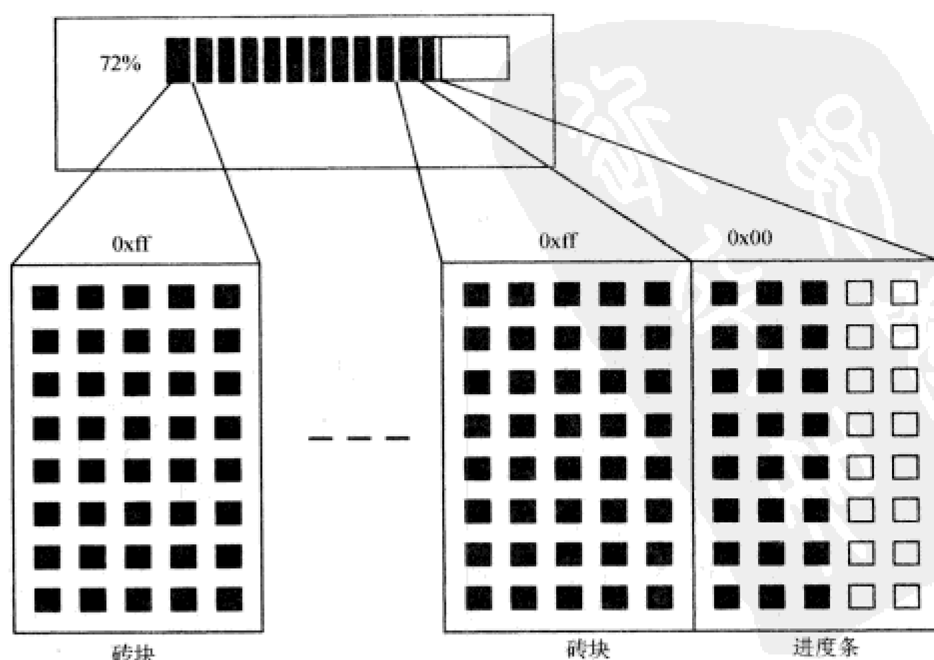


图 10-8　绘制进度条

下面是定义一个具有特定厚度的进度条末梢的代码。

```
void newBarTip( int i, int width)
{
    char bar;
    int pos;

    // save cursor position
    while( busyLCD());
    pos = addrLCD();

    // generate a new character at position i
    // set the data pointer to the LCD CGRAM buffer
    setLCDG( i*8);

    // as a horizontal bar (0-4)x thick moving left to right
    // 7 pixel tall
    if ( width > 4)
      width = 0;
    else
      width = 4 - width;

    for( bar=0xff; width > 0; width--)
      bar<<=1;              // bar >>= 1; if right to left

    // fill each row (8) with the same pattern
    putLCD( bar);
    putLCD( bar);
    putLCD( bar);
    putLCD( bar);
    putLCD( bar);
    putLCD( bar);
    putLCD( bar);
    putLCD( bar);
    // restore cursor position
    setLCDC( pos);
} // newBarTip
```

具备了这些必需的材料，绘制一个实际的进度条就只需再添加几行代码：

```
void drawProgressBar( int index, int imax, int size)
{   // index is the current progress value
    // imax is the maximum value
    // size is the number of character positions available
    int i;

    // scale the input values in the available space
    int width=index * (size*5)/imax;

    // generate a character to represent the tip
    newBarTip( TIP, width % 5); // user defined character 0

    // draw a bar of solid blocks
    for ( i=width/5; i>0; i--)
          putLCD( BRICK); // filled block character

    // draw the tip of the bar
    putLCD( TIP); // use character 0

} // drawProgressBar
```

可见，为了使 drawProgressBar() 函数真正地好用，输入值被缩放了，以便进度条能够适应 LCD 显示屏上提供的空间，并且根据给定的最大值得出进度值，再作为参数传递下去。为了测试这段代码，需要定义一个循环，其中计数值（index）从 0 至 99 缓慢地循环变化。显示屏第一行的前 3 个字符用于显示进度值，而其他空间用于放置进度条。

```
main( void)
{
    int index;
    char s[8];

    // LCD initialization
    initLCD();

    index = 0;

    // main loop
    while( 1)
    {
      clrLCD();

      sprintf( s, "%2d", index);
      putsLCD( s); putLCD( '%');

      // draw bar
      drawProgressBar( index, 100, HLCD-3);

      // advance and keep index in boundary
      index++;
      if ( index > 99)
      index=0;

      // slow down the action
      Delayms( 100);

    } // main loop
} // main
```

请注意，通过插入一个小延时来减慢主循环的执行速度很关键，否则显示屏的刷新速度太快，看到的就是鬼影似的模糊景象。请记住，LCD 显示屏属于慢速设备，对它们要有耐心！

最后，在开始生成工程前，请记得添加所有需要使用的函数库。请选择工程窗口，然后用单击源文件，选择 Add file 选项。浏览到刚创建的 *lib* 目录，然后选择 explore.c 文件（它提供 Delayms() 函数）以及 LCDlib.c 文件。

现在生成工程，然后用调试器对 Explorer 16 演示板编程，并观察代码运行情况。LCD 屏幕上的进度条正光滑地从左向右移动，并填满整行。这真是令人乐不可支啊！

10.13　小结

本章介绍了如何使用并行主端口与字符式 LCD 显示模块相连。LCD 显示模块只是众多需要 8 位并行接口的常见设备之一。由于 LCD 显示模块属于相对较慢的外围设备，因此你可能会觉得使用 PMP 与传统的“位脉冲”（bit-banged）方法相比并没有什么优势（或者优势不明显）。实际上，即使是访问这种简单而缓慢的外围设备，使用 PMP 仍然具有两个重要优点。

- 始终要确保控制信号的时序、顺序和复用能和配置参数相匹配，从而避免出现危险的总线冲突或不稳定的操作。这些问题会导致编码错误或者不期望的操作和时序状态（中断、bug 等）。

- MCU 完全不必照看外部（外围设备）总线，从而能并行执行不打断接口时序的各种高优先级任务。

10.14 对 PIC24 单片机行家的提示

PIC32 的 PMP 模块与 PIC24 的几乎完全一样，只是增加了一些重要功能。下面是一些会影响 PIC32 程序移植的主要区别。

（1）PMCON 寄存器控制位的布局有变化，变得与大多数其他外围设备类似，因此 ON、FRZ 和 IDL 位现在都位于标准位置（分别是 15 位、14 位、13 位）。

（2）删除了 PMBE 的输出信号。

（3）PMPTTL 控制位现在位于 PMCON 寄存器中，用于选择施密特触发或者 TTL 输入电平。而过去在 PIC24 中它位于 PADCFG1 寄存器中。

（4）修改了 PMMODE 寄存器的 IRQM=11 和 IRQM=10 选项。

（5）PMPEN 寄存器改名为 PMAEN。最新的 PIC24 数据手册也更新为类似的名字。

（6）提供单独的 PMDIN 和单独的 PMDOUT 寄存器（现在是 32 位宽），从而可以同时访问所有的数据缓存。

10.15 提示与技巧

尽管基本的字符式显示屏已经围绕 HD44780 控制器接口和命令集进行了很好的标准化，但是图形式显示屏则与此大不相同。目前市面上存在各种控制器，它们的功能也各不相同。小型 LCD 显示屏最常采用的控制器是 New Japan Radio 公司的 NJU6679，它用在很多单色显示屏上（最大尺寸达 128×128），采用与 HD44870 类似的并行接口。但是最近的趋势是以爱普生公司的 S1D15G10 为代表的串行接口控制器，它用于很多廉价的彩色 LCD 显示屏（最常见的就是为黑白屏诺基亚手机所另加的彩屏），据说最新一代的 3G 手机要大量使用它，因而其价格低廉。OLED 显示屏也采用串行接口（SPI）。而当显示屏的分辨率超过 QVGA（320*240）时，就不能指望在显示屏上找到完整的控制器芯片了，必须产生复杂的同步信号来连续刷新屏幕，此时就必须配备 QVGA 或者更先进的显示外围设备电路。

10.16 练习

（1）和前面使用异步串行接口的各章中建议的一样，可以重新定义 stdio.h 库函数的输出，比如用 printf() 向 LCD 显示屏输出。请重新定义_mon_putc() 函数（详细内容请参阅 *MPLAB C32 C Library Guide*）通过并行主控接口向 LCD 发送字符。

（2）LCD 显示屏通常是非常缓慢的设备。当 PIC32 等待 LCD 显示屏执行命令时会浪费很多处理器时间。请利用缓存原理和定时器中断实现一个后台 LCD 显示接口（PIC24 和 dsPIC 平台的 Explorer 16 演示板提供的 LCD.c 文件中有这种原理的基本示例）。

10.17 参考书

Jeremy Bentham 所著的《嵌入式系统 Web 服务器：TCP/IP Lean》。这本书会将你带入难度更高的级别，阐述如何用几行 C 代码实现因特网的基础——TCP/IP 协议。作者深谙简化之道，这是每个嵌入式控制应用系统必需的。

10.18 链接

www.microchip.com/graphics。Microchip 公司提供的图形函数库支持大多数流行的 16 位和 32 位 LCD 显示控制器。你可以在 Web Graphic Design Center 上获得免费版和第三方提供的函数库。

www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1824&appnote=en011993。这是 Microchip 公司的使用说明书 AN833 的链接，免费提供适用于所有 PIC 单片机的 TCP/IP 协议程序集。

www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1824&appnote=en012108。使用说明书 AN870 介绍了一种适用于基于 Microchip TCP/IP 协议程序集的应用系统的简单网络管理协议。

第 11 章 模数转换

11.1 计划

我们生活在模拟的世界里。温度、湿度、压力以及电压、电流都是模拟量。如果希望嵌入式控制系统能与外界相互作用，那么就必须学会理解模拟信息并将它们转换为数字信号，以便单片机进行加工，并可能再次产生模拟输出值。模-数转换模块是嵌入式控制系统与“真实”世界接口的关键设备之一。PIC32MX 系列单片机是针对嵌入式控制应用系统设计的，因此也是处理模拟信号的理想选择。该系列单片机都集成有高速 ADC（Analog-to-Digital Converter，模-数转换器），它每秒能完成 500 000 次转换，并带有输入多路复用器，从而能够快速、高分辨率地监控多路模拟输入。本章将学习如何用 PIC32MX 系列单片机的 10 位 ADC 在 Explorer 16 演示板上实现简单的测量：首先读取电位计上的电压，然后读取温度传感器的输入。

11.2 准备

本章除了需要通常的软件工具，如 MPLAB IDE、MPLAB C32 编译器、MPLAB SIM 仿真器之外，还要使用 Explorer 16 演示板以及你所选择的在线调试器。

11.3 探索

和 PIC32 上的其他外围设备模块一样，使用 ADC 的第一步是熟悉该模块的结构以及关键的控制寄存器。

是的，这意味着你需要再次阅读器件的数据手册，甚至还要查阅 Explorer 16 用户指南中的电路原理图。

首先，请看 ADC 单元的框图（参见图 11-1）。

该 ADC 的架构十分复杂，具备很多有趣的功能。

- 多达 16 路模拟输入。
- 两个输入多路复用器，每个都可用于选择不同的输入模拟通道以及不同的参考电压源。
- 10 位转换器的输出可以格式化为整数或者定点数、有符号数或者无符号数、16 位数或者 32 位数。
- 控制逻辑提供很多自动转换方式，可以使转换过程和其他相关的模块及输入引脚的动作同步。
- 转换输出存储在 32 位宽、16 字深的缓存中，该存储区可以配置成顺序扫描模式，或者作为简单的 FIFO 缓冲区。

上述所有功能都需要合理配置大量的控制寄存器。我知道，读者面对如此众多的选项并要考虑如何选择是件令人头昏的事情（特别是在最开始）。因此，我们首先要以最直接最简单的方式完成一个最简单的示例应用：读取 Explorer 16 板上电位计 R_6的电压，见图 11-2。其中，10kΩ 的电位计与电源轨直接连接，因此其输出可涵盖 3.3V 至参考地之间的范围。R_6 的输出与单片机的 RB_5 引脚相接，该引脚对应于 ADC 输入多路复用器的输入 AN_5。

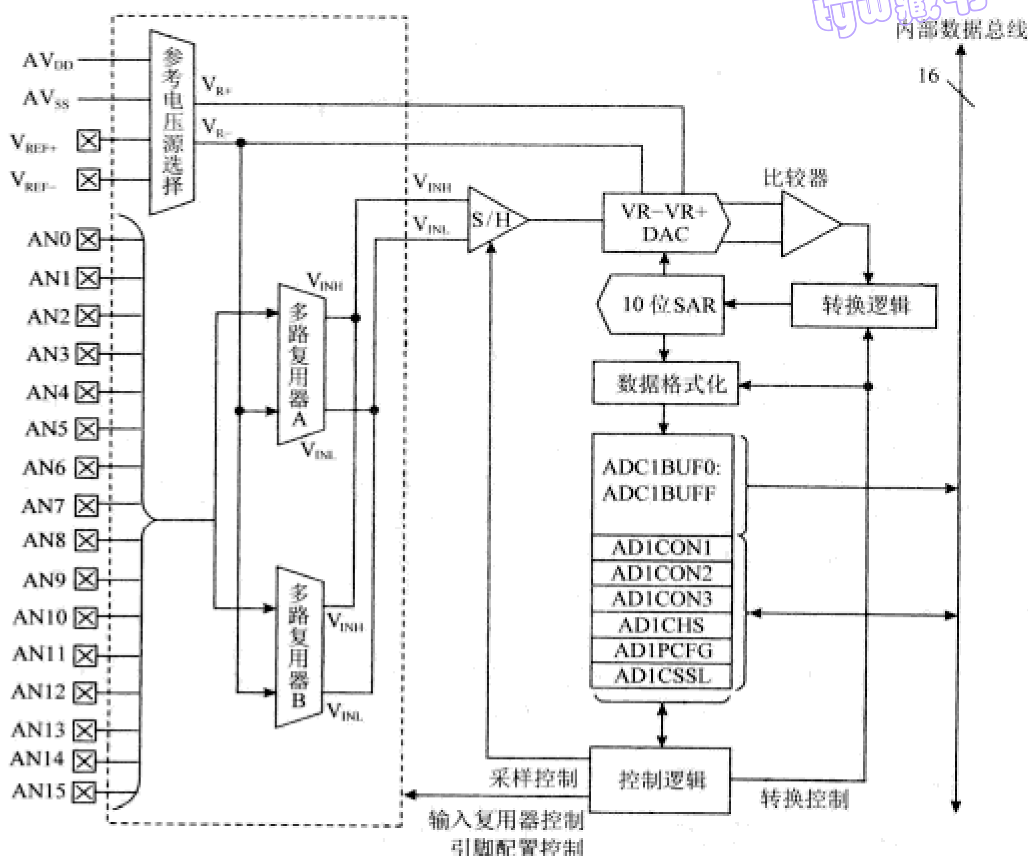


图 11-1　10 位高速 ADC 的组成框图

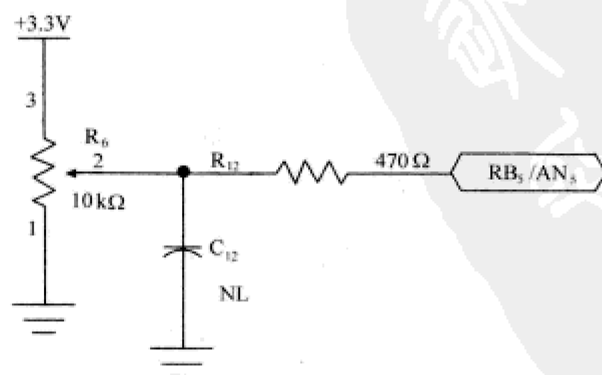


图 11-2　Explorer 16 板上电位计 R_6 的详细电路图

利用检查表创建新工程后，新建一个源文件 pot.c，引用常用的头文件以及一些有用的常数定义。第一个常数是 POT，它定义了电位计使用的输入通道；第二个是屏蔽字 AINPUTS，它用于将输入引脚定义成模拟的或数字的：

```
/*
** It's an analog world
** Converting the analog signal from a potentiometer
*/
// configuration bit settings, Fcy=72 MHz, Fpb=36 MHz
#pragma config POSCMOD=XT, FNOSC=PRIPLL
#pragma config FPLLIDIV=DIV_2, FPLLMUL=MUL_18, FPLLODIV=DIV_1
#pragma config FPBDIV=DIV_2, FWDTEN=OFF, CP=OFF, BWP=OFF
#include <p32xxxx.h>

#define POT 5 // 10 k potentiometer on AN5 input
#define AINPUTS 0xffef // Analog inputs POT, TSENS
```

所有 ADC 控制寄存器的初始化工作最好在一个简短的函数 initADC()中进行，它负责完成初始配置。

- AD1PCFG 用于屏蔽模拟输入通道选择：0 表示将对应引脚配置为模拟输入，1 则配置为数字输入。
- AD1CON1 设置自动开始转换，可以在自动定时采样阶段结束时触发。此外，输出值被格式化为简单的无符号数或右对齐（整型）数。
- 由于不使用扫描函数（只有一路输入），AD1CSSL 将被清零。
- AD1CON2 选择使用多路复用器 A，将 ADC 参考输入连接到模拟输入电源轨 AVdd 和 AVss 引脚。
- AD1CON3 选择转换时钟源和分频器。
- 最后配置 ADON，整个 ADC 外围设备都会被激活并做好使用准备。

```
void initADC( int amask)
{
  AD1PCFG = amask;        // select analog input pins
  AD1CON1 = 0;            // manual conversion sequence control
  AD1CSSL = 0;            // no scanning required
  AD1CON2 = 0;            // use MUXA, AVss/AVdd used as Vref+/-
  AD1CON3=0x1F02;        // Tad=2+1) x 2 x Tpb=6x27 ns>75 ns
  AD1CON1bits.ADON=1;    // turn on the ADC
} //initADC
```

amask 可看作初始化子程序的参数，这样处理就可以在今后的应用中灵活地选择不同的（多路）输入通道。

> **注解** 和 PIC32 内置的其他外围设备模块一样，ADC 模块对应的外围设备函数库会提供一组函数和宏定义，能够简化代码，或者至少使访问 ADC 模块的代码更具可读性。鉴于 ADC 模块的灵活性，我个人建议你第一次最好通过直接访问各种控制寄存器来熟悉 ADC 运行时的底层细节，不要依赖外围设备函数库的帮助。

11.4 完成第一次转换

实际的模-数转换过程分为两步。首先要采样输入电压信号；然后断开输入引脚并开始真正的转换过程，将已采样的模拟电压信号转换成数字信号。这两个阶段由 AD1CON1 寄存器中的两个独立的控制位 SAMP 和 DONE 进行控制。这两个阶段的时序对测量结果的准确度有重要影响。

- ❑ 在采样阶段，外部信号与内部电容相连，该电容会被充电并接近输入电压。充电时间必须足够长，以确保电容能跟踪输入电压，并且该充电时间与输入信号源的阻抗（在本例中，阻抗等于 10kΩ）以及内部电容的大小成比例。通常，在能与输入信号频率匹配的情况下（这里不讨论该问题），采样时间越长，结果越好。
- ❑ 转换阶段的时序与所选的 ADC 时钟源有关。该时钟源可以是外部总线时钟信号的分频信号，也可以是单独的 RC 振荡器。别看 RC 振荡器的结构很简单，当 PIC32 需要在低功耗模式且外围设备时钟关闭的情况下进行模数转换时，它就是很好的选择。然而，在其他大多数情况下，使用振荡器时钟分频则是更好的选择，这是因为它能与外围设备总线的运行同步，从而能更好地抑制内部噪声。在符合 ADC 模块规范要求的情况下，转换时钟要尽可能快。

下面是基本的转换例程。

```
int readADC( int ch)
{
  AD1CHSbits.CH0SA = ch;       // 1. select analog input
  AD1CON1bits.SAMP = 1;        // 2. start sampling
  T1CON = 0xs8000; TMR1 = 0;   // 3. wait for sampling time
  while (TMR1 < 100);          //
  AD1CON1bits.SAMP = 0;        // 4. start the conversion
  while (!AD1CON1bits.DONE);   // 5. wait conversion complete
  return ADC1BUF0;             // 6. read result
} // readADC
```

11.5　自动采样的时序

可见，我们采用最基本的方法，即利用定时器实现两个等待循环，就为采样阶段提供了精确的时序。在 PIC32 的 ADC 模块中，采样阶段可以自定义为最多 $32 \times T_{ad}$。是否采用该功能仅取决于源阻抗与 ADC 输入电容的乘积。只要将 AD1CON1 寄存器的 SSRC 位设置为 0x7，就能在自定时采样周期结束后自动启动转换。采样周期本身的长度是由 AD1CON3 寄存器的 SAM 位控制的。下面是改进后的新程序，它采用了自动定时采样与转换触发器：

```
void initADC( int amask)
{
  AD1PCFG = amask;          // select analog input pins
  AD1CON1 = 0x00E0;         // automatic conversion after sampling
  AD1CSSL = 0;              // no scanning required
  AD1CON2 = 0;              // use MUXA, use AVdd & AVss as Vref+/-
  AD1CON3 = 0x1F3F;         // Tsamp = 32 x Tad;
  AD1CON1bits.ADON = 1;     // turn on the ADC
} //initADC
```

请注意是如何产生启动转换：它是在自定时采样阶段结束时自动触发的。这样我们就可以做到：

- ❑ 无需使用定时延时循环和其他定时源就能保证采样阶段的时间合适；
- ❑ 一条命令（启动采样阶段）就能完成整个采样与转换过程。

ADC 经过如此配置后，启动转换和读取转换结果就变得很简单了。

- ❑ 用 AD1CHS 选择来自多路复用器 A 的输入通道。
- ❑ 置位 AD1CON1 寄存器的 SAMP 位，启动定时采样，之后立即开始转换。
- ❑ 当整个过程结束并且转换结果就绪后，AD1CON1 寄存器的 DONE 位置位。
- ❑ 读取 ADC1BUF0 寄存器，立即得到期望的转换结果。

```
int readADC( int ch)
{
  AD1CHSbits.CH0SA = ch;       // 1. select input channel
  AD1CON1bits.SAMP = 1;        // 2. start sampling
  while (!AD1CON1bits.DONE);   // 3. wait conversion complete
  return ADC1BUF0;             // 4. read conversion result
} // readADC
```

11.6 开发演示系统

现在还要做的是找出一种有趣的方式，在 Explorer 16 演示板上演示转换结果。使用接在 PORTA 上的 LED 灯是一种令人感兴趣的方式，但是那些使用 PIC32 Starter Kit 的用户就无法体验这种快乐，因为这种开发板上的大部分 PORTA 引脚都与 JTAG 端口相接。因此，我们将使用第 10 章开发的 LCD 函数库来显示块状条形图。没错，我们要使用第 10 章开发的漂亮且光滑的进度条，但是希望你不会被这些细节分散精力。下面是我们将用来测试模-数转换功能的主程序：

```
main ()
{
  int i, a;

  // initializations
  initADC( AINPUTS); // initialize the ADC
  initLCD();// initialize the LCD display

  // main loop
  while( 1)
  {
    a = readADC( POT); // select the POT input and convert

    // reduce the 10-bit result to a 4 bit value (0..15)
    // (divide by 64 or shift right 6 times
    a >> = 6;

    // draw a bar on the display
    clrLCD();
    for ( i=0; i<=a; i++)
      putLCD( 0xFF);

    // slow down to avoid flickering
    Delayms( 200);
  } // main loop
} // main
```

调用 ADC 初始化子程序后，就该初始化 LCD 显示模块了。然后在主循环中对 AN5 进行转换并且将输出结果格式化为适合特殊显示需要的形式。根据上述代码的配置，10 位转换输出将变成范围在 0～1023 之内的右对齐整数。再把结果除以 64（或者说将它右移 6 位），就可以将结果减小为 0~15 之内的数。显示与结果个数相同的方块，就能得到一个长度与电位计输出电压成比例的方块条。

请记得将#include<>语句添加到 LCD.h 函数库并将 lib 目录下的 explore.c 和 LCDlib.c 添加到工程的源文件列表中。

生成工程，然后用通用的 In Circuit Debugging（在线调试）检查表对 Explorer 16 演示板编程。如果一切正常，你就能使用电位计了，把电位计从一侧滑动到另一侧就能观察到 16 个方块对应地从左侧移向右侧。

11.7　创建自己的小型 ADC 函数库

我们将反复使用初始化 ADC 模块和完成单次自定时转换的简单子程序。为此，可以将它们单独放在小型函数库 ADClib.c 中，并将这个新函数库添加到 lib 文件夹中。

```
/*
** ADClib.c
**
*/
#include <p32xxxx.h>
#include <ADC.h>

// initialize the ADC for single conversion, select input pins
void initADC( int amask)
{
  AD1PCFG = amask;              // select analog input pins
  AD1CON1 = 0x00E0;             // auto convert after end of sampling
  AD1CSSL = 0;                  // no scanning required
  AD1CON2 = 0;                  // use MUXA, AVss/AVdd used as Vref+/-
  AD1CON3 = 0x1F3F;             // max sample time = 31Tad
  AD1CON1SET = 0x8000;          // turn on the ADC
} //initADC

int readADC( int ch)
{
  AD1CHSbits.CH0SA = ch;        // select analog input channel
  AD1CON1bits.SAMP = 1;         // start sampling
  while (!AD1CON1bits.DONE);    // wait to complete conversion
  return ADC1BUF0;              // read the conversion result
} // readADC
```

和 LCD.h 文件一样，我们可以将基本的定义集以及用于访问库函数的原型单独放在一个文件中，并将它保存到 include 目录中。

```
/*
** ADC.h
**
*/
#define POT 5             //10k potentiometer on AN5 input
#define TSENS 4           // TC1047 Temperature sensor on AN4
#define AINPUTS 0xffcf    // Analog inputs for POT and TSENS

// initialize the ADC for single conversion, select input pins
void initADC( int amask) ;
int readADC( int ch);
```

这样就很简单了。接下来还有更多的乐趣和游戏！

11.8　乐趣与游戏

我得承认，上个工程并不是很令人兴奋。毕竟，我们使用的是 32 位单片机，它的时钟频率达 72MHz，集成的 10 位模-数转换器每秒能完成几十万次转换。但是我们只保留了转换结果中的 4 位数据，并且在 LCD 显示屏上只能看到一个方块移动。能否做得更具挑战性并且更好玩呢？开发一个一维的 Pac-Man 游戏[①]。我们是否应该叫它“Pot-Man”游戏？

---

①即 20 世纪 80 年代非常流行的“吃豆人”或“小精灵”电子游戏。——译者注

在这个古老的 Pac-Man 游戏（别告诉我你没听说过这个游戏，但是如果真的没听过，请查维基百科）里，小精灵 Pac 在二维的迷宫里走来走去，绝望地寻找食物。现在，只需稍加修改，我们就能构建出一个该游戏的一维简化版，并根据移动方向用<或者>字符表示 Pac。Pac 只能在 LCD 显示屏的某一行上左右移动，并受电位计位置的控制。而食物块则由*字符表示，并且随机地放置在与 Pac 同行上的某个位置。一旦 Pac 到达某个食物所在处，就能吞下食物并且继续移动，然后在其他位置又会出现一个新食物。

这里将再次使用非常重要的伪随机数发生器函数 rand()（在 stdlib.h 中定义）。所有的游戏都需要一定的不可预测性，而伪随机发生器正是电脑游戏在逻辑世界实现的一种方式，并且不会无限重复。

首先，要修改前面的工程代码或者重新编写一个全新的 Pot-Man.c 文件。然后创建一个新工程，我建议就将它命名为 POT。实际上，只需添加几行代码就能实现简单的动画。

```
/*
** Pot-Man.c
**
*/
// configuration bit settings, Fcy=72MHz, Fpb=36MHz
#pragma config POSCMOD=XT, FNOSC=PRIPLL
#pragma config FPLLIDIV=DIV_2, FPLLMUL=MUL_18, FPLLODIV=DIV_1
#pragma config FPBDIV=DIV_2, FWDTEN=OFF, CP=OFF, BWP=OFF

#include <p32xxxx.h>
#include <explore.h>
#include <LCD.h>
#include <ADC.h>

main ()
{
  int a, r, p, n;
  // 1. initializations
  initLCD();
  initADC( AINPUTS);

  // 2. use the first reading to randomize
  srand( readADC( POT));

  // 3. init the hungry Pac
  p = '<';

  // 4. generate the first random food bit position
  r = rand() % 16;

  // main loop
  while( 1)
  {

    // 5. select the POT input and convert
    a = readADC( POT);

    // 6. reduce the 10-bit result to a 4 bit value (0..15)
    // (divide by 64 or shift right 6 times
    a>> = 6;

    // 7. turn the Pac in the direction of movement
    if ( a < n)// moving to the left
      p = '>';
    if ( a > n)// moving to the right
```

```
    p = '<';

  // 8. when the Pac eats the food, generate more food
  while (a == r )
    r=rand() % 16;

  // 9. update display
  clrLCD();
  setLCDC( a); putLCD( p);
  setLCDC( r); putLCD( '*');

  // 10. provide timing and relative position
  Delayms( 200);// limit game speed
  n=a;          // memorize previous position
 } // main loop
} // main
```

- 第 1 部分，对 ADC 模块和 LCD 显示模块进行常规的初始化。
- 第 2 部分，首次读取电位计值，并以电位计的位置作为伪随机数发生器的种子。这样可以使每次的游戏情况不同。但是要保证电位计不会总在最左侧或最右侧。因为那样对应的种子值就总为 0 或者 1023，这导致每次游戏重启时伪随机数都会以完全相同的顺序出现，游戏会因此而变得重复。
- 第 3 部分，可以指定小精灵 Pac 的初始方向为任一方向。
- 第 4 部分，确定第一块食物的第一个随机位置。
- 第 5 部分，在主循环内检查电位计滑片的最新位置。
- 第 6 部分，只保留 10 位整数的高四位，对应 0～15。
- 第 7 部分，比较新位置和前一次循环检测到的位置，决定 Pac 该向哪个方向移动。如果 ADC 结果减小，那就意味着电位计发生逆时针转动。因此 Pac 要向左移动。反之，如果 ADC 结果与上一次循环的结果相比增大了，那就意味着电位计发生顺时针转动，因此 Pac 要向右移动。
- 第 8 部分，比较 Pac 的新位置（ADC 读数）与食物的位置，如果这两个位置重合（即 Pac 抵达食物处），那么就立刻计算新食物的随机位置（r）。由于新位置可能与旧位置相同（如果所用的伪随机数发生器很好，那么概率为 1/16），因此这个过程要在 while 循环中完成。换句话说，我们创建的新“玉米粒”可能就在 Pac 的嘴边。我们显然不愿意这样，难道你不觉得这样太没挑战性了么？
- 最后，第 9 部分，清除显示内容，然后在新位置放置代表 Pac 和食物的两个符号。
- 第 10 部分，以一个短暂的延时结束循环，并保存 Pac 的位置以便下一次循环比较。

别忘记引用工程 lib 目录下的 LCDlib.c、ADClib.c 以及 Explore.c 文件。生成工程，并将它烧录到 Explorer 16 演示板上。你不得不承认：模-数转换现在变得有趣多了！

11.9　温度检测

接下来的任务更困难，Explorer 16 演示板上安装有温度传感器，它正巧是 Microchip 公司的高线性度电压输出型集成温度传感器件 TC1047A。该器件非常小，采用 SOT-23（3 个引脚、贴片安装）封装，功耗不超过 35μA（典型值），工作电压范围是 2.5～5.5V。它的输出电压与电源电压无关，并且和温度成严格的线性关系（通常在 0.5℃），斜率为 10mV/℃。其偏置电压与绝对温度的关系参见图 11-3 中的公式。

在此，我们可以再次使用 PIC32 的 ADC 将模拟输出转换为数字信号。根据 Explorer 16 演

示板的原理图（参见图 11-4），温度传感器的输出直接与模拟输入通道 AN4 相连。

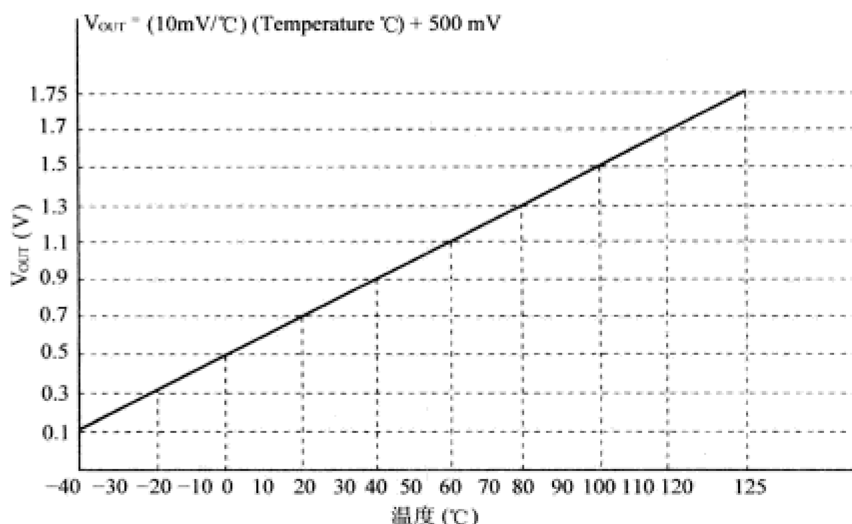


图 11-3 TC1047A 的输出电压与温度的关系特性

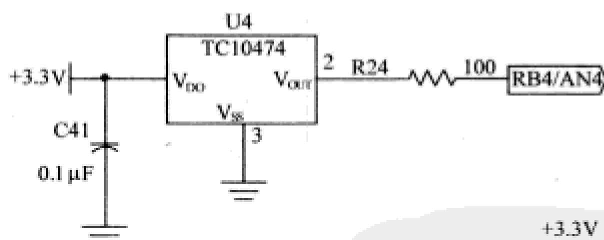


图 11-4 Explorer 16 演示板的温度传感器 TC1047A 的详细电路图

我们可以再次使用前面的示例中开发的 ADC 函数库，并把它放进新建的工程 TEMP 中，把前面的代码保存为 Temp.c。

下面要修改代码，增加一个新的常数定义 TSENS，将它作为与温度传感器相接的 ADC 输入通道。

```
/*
** Temp.c
** Converting the analog signal from a TC1047 Temp Sensor
*/
// configuration bit settings, Fcy=72 MHz, Fpb=36 MHz
#pragma config POSCMOD=XT, FNOSC=PRIPLL
#pragma config FPLLIDIV=DIV_2, FPLLMUL=MUL_18, FPLLODIV=DIV_1
#pragma config FPBDIV=DIV_2, FWDTEN=OFF, CP=OFF, BWP=OFF

#include <p32xxxx.h>
#include <explore.h>
#include <LCD.h>
#include <ADC.h>
```

可见，ADC 配置或者启动转换部分无需任何修改。但是在 LCD 上显示结果可能有些麻烦。温度传感器本身带有一定的噪声，为了使读数更加稳定，通常需要进行一些滤波。比如，将 1s

周期内的 10 个采样值分为一组，取其平均就能得到更加稳定的读数：

```
a=0;
  for ( j=0; j < 10; j++)
    a+=readADC( TSENS);      // add up 10 readings
  i=a/10;       // divide by 10 to average
```

根据图 11-3 中的公式，可以计算出 Explorer 16 演示板上的 TC1047A 测得的绝对温度。事实上，可以根据下面的公式求出温度值：

$$T = \frac{V_{out} - 500\,\text{mV}}{10\,\text{mV/C}}$$

其中：

$$V\text{out} = \text{ADC 的读数} \times \text{ADC 的分辨率（mV/bit）}$$

由于 PIC32 的 ADC 单元配置成使用与 Vdd（3.3V）相连的 AVdd 作为内部参考电压源，而已知 ADC 是 10 位的，于是可知 ADC 的分辨率为 3.3mV/bit。因此，温度值也可以表示为：

$$T = \frac{3.3i - 500}{10}$$

我们可以很容易地在 LCD 显示屏上显示绝对温度结果，但是这样岂不是很没意思？要是用一个字符的位置作为指针指示相对温度怎么样？或者，用温度值控制前面工程开发的一维 Pac-Man 游戏怎么样？我们可以通过向传感器吹热气对其加热，从而使 Pac 向右移动，并通过向传感器吹冷气使 Pac 向左移动。

从实际的角度看，这很容易实现。可以在进入主循环前采样初始温度值，然后把它作为参考值以决定 Pac 相对显示中心的位置偏移量。在主循环中更新光标的位置，当温度升高时向右移动，或者当检测到温度降低时向左移动。下面是新的 Temp-Man 游戏的完整代码，也可以把它称为呼吸分析仪游戏。

```
main ()
{
  int a, i, j, n, r, p;

  // 1. initializations
  initADC( AINPUTS); // initialize the ADC
  initLCD();

  // 2. use the first reading to randomize
  srand( readADC( TSENS));
  // generate the first random position
  r = rand() % 16;
  p = '<';

  // 3. compute the average value for the initial reference
  a = 0;
  for ( j=0; j<10; j++)
  {
    a+=readADC( TSENS); // read the temperature
    Delayms( 100);
  }
  i=a/10; // average

  // main loop
  while( 1)
  {
```

```
// 4. take the average value over 1 second
a = 0;
for ( j=0; j<10; j++)
 {
   a += readADC( TSENS); // read the temperature
   Delayms( 100);
 }
 a /= 10;       // average result

 // 5. compare initial reading, move the Pac
 a=7+(a-i);

 // 6. keep the result in the value range 0..15
 if ( a > 15)
   a = 15;
 if ( a < 0)
   a = 0;

 // 7. turn the Pac in the direction of movement
 if ( a < n)// moving to the left
   p = '>';
 if ( a > n)// moving to the right
   p = '<';

 // 8. as soon as the Pac eats the food, generate new
 while (a == r )
   r = rand() % 16;

 // 9. update display
 clrLCD();
 setLCDC( r); putLCD( '*');
 setLCDC( a); putLCD( p);

 // 10. remember previous postion
 n = a;

} // main loop
} // main
```

你会发现，其中的大部分代码与前面的工程/游戏完全一样。而它们的显著区别主要在以下几个部分。

- 第 3 和第 4 部分，用 1s 周期内 10 次采样值的平均代替单次采样值。
- 第 5 部分，计算温度差，并将此作为相对于中心位置（7）的偏移量。
- 第 6 部分，检查边界。如果偏差值变为负数并且超过 4 位宽度，就在最左侧显示。当偏差值为正数并且超出 4 位宽度时，在最右侧显示。
- 第 10 部分，由于读取温度值并计算平均值过程已经使游戏速度正常，因此不需要再延时。

请记得包含需要使用的所有函数库，并根据常用的检查表生成该工程。然后用所选的在线调试器将程序烧录到 Explorer 16 演示板上，试试看。

你遇到的第一个问题可能是如何找出板上极小的温度传感器。（提示：它非常靠近处理器单元的左下角，看起来像贴片三极管。）第二个立刻面临的问题则是如何向电路板吹气，产生热空气和冷空气从而使 Pac 动起来。这看起来容易，其实很难。事实上，我个人觉得吹冷风最难；有些朋友说这可能和我当时所处的环境有关，如果我在市场里工作，那么到处都是热空气！

11.10 小结

本章只接触到一些皮毛知识，并尝试用 PIC32 的 ADC 模块探索模拟世界。我们使用了众多配置中的一种简单配置，并且只用到少量高级功能。我们还尝试获取 Explorer 16 演示板提供的两种模拟输入信号，希望你能在此过程中获得乐趣。

11.11 对 PIC24 行家的提示

PIC32 的 ADC 模块与 PIC24 的基本一样，但还是增加了一些重要功能。下面是一些会影响 PIC32 程序移植的主要区别。

（1）在 AD1CON1 寄存器中，转换格式选项如今已经扩展为 32 位小数形式。

（2）CLRASAM 控制位如今添加到 AD1CON1 寄存器中，它能在第一次中断后停止转换过程。

（3）AD1CON2 寄存器新增了自动标定模式，用于减小 ADC 偏置。新增 OFFCAL 控制位，用于进入标定模式。

（4）AD1CHS 寄存器的控制位如今位于 32 位字的上半部分。还有一个 CH0NB0 控制位，它用于选择第二个输入多路复用器的负极性输入。

11.12 提示与技巧

如果所需的采样时间超出能够提供的最大时间（$32 \times T_{ad}$），那么可以先尝试扩展 T_{ad}，更好的方法则是重新开始并允许自动开始采样（在结束转换时）。这样无论是否开始转换，采样电路始终打开并充电。此外，利用定时器 3 周期性地清除 SAMP 位（AD1CON1 中 SSRC 位的功能之一）并且允许 ADC 转换结束中断提供更宽的采样周期，同时使所需的 MCU 开销最小。在这种情况下，没有无限等待循环，只有当得到转换结果并准备被提取时，才产生周期性的中断。

此外，并不是所有的应用系统都需要对模拟输入量进行完全转换。PIC32MX 系列单片机还提供（两个）模拟比较模块，并带有专用的输入多路复用器。它们可用于那些在模拟输入量超出阈值时需要做出快速响应的应用。比较时不需要配置 ADC、选择通道和转换，并且能连续进行。当达到参考电压时就会立即产生中断（或者产生一个输出信号）。

说到参考电压，还有另一个称为比较器参考（Comparator Reference）的模块可以使用，该模块能有效地表示某一类数-模转换器。它可以和比较器模块一起使用，也可以单独使用，产生多达 32 个参考电压。

11.13 练习

（1）利用 ADC FIFO 缓存收集转换结果，并配置定时器 3 实现自动转换和中断，当该缓存满时才调用函数对结果进行平均。

（2）试验与其他模拟传感器接口（利用 Explorer 16 板的原型区），比如压力传感器、湿度传感器甚至加速度计。两坐标或三坐标固态加速度计的价格正在逐步降低，很容易获得。与它们相接只需要一些模拟输入引脚和一个快速的 10 位 ADC 模块。

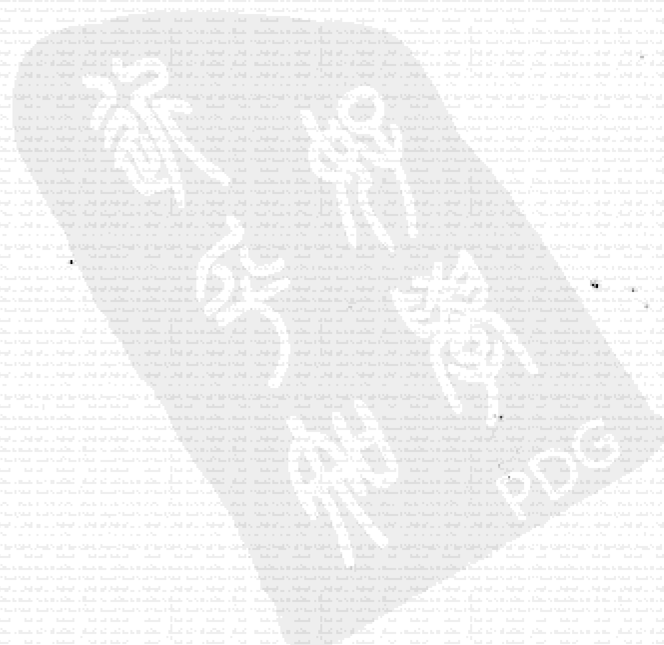
11.14 参考书

Bonnie Baker 所著的 *A Baker's Dozen: Real Analog Solutions for Digital Designer*。介绍如何合理周到地使用模-数转换器最详尽的参考书。

11.15 链接

www.microchip.com/filterlab。可以从该网址下载免费的 FilterLab 软件；它能帮助你快速高效地设计出适合模拟输入的抗混叠滤波器。

www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=2102&param=en021419&pageId=79。这里有各种温度传感器以及不同的接口选择，包括直接的 I^2C 或 SPI 数字输出。


PDG

第三部分

扩　展

恭喜你！你又完成了5章的学习！你已经学会了使用PIC32MX的一些关键硬件外围设备模块，还在Explorer 16演示板上进行了实践。

在本书的第三部分，我们将开发几个新工程，这些新工程需要你能很快学会使用几个新模块。因为这些工程的难度会有所增加，所以你最好在手边准备一个实际的演示板（Explorer 16），同时也需要你能够通过局部的改动和利用原型板区来增加演示板的功能。在以下章节中，会在需要时给出简单的原理图和元器件编号。在本书配套网站www.ExploringPIC32.com上，你可以找到更多关于扩展板和原型板区的功能说明，它们可以帮助你更好地开发更高级的工程。

第 12 章　捕获用户输入

12.1　计划

按理说模拟输入信号才是嵌入式控制程序和外部世界间接口的关键所在，但一直以来建立用户接口的真正基础却是数字输入信号。和这个一样错位的还有，在很长一段时间内，人类的思维模式也被训练成只采用按钮和开关的方式来和机器进行交互。这是因为采用语音、手势和视听这些方式会使人机接口的复杂性增加数倍，以至于人们宁可使用按钮和开关，从而能通过简单的“是”（1）和“不是”（0）这种本原的方式来和机器进行交互。而最近由视频游戏和移动电话厂商所发起的一些革新，正是由于采用了更为复杂的交互方式，因而产生了很高的关注度，引起了消费者的狂热追捧。例如任天堂 Wii 的加速度传感器，还有 iPhone 的多触点触摸感应屏。

在本章中，我们将探索用多种方式来捕获“传统”的用户输入。通过检测按钮和简单的机械开关的动作、读取旋转编码器上的输入以及读取计算机键盘可以得到这些用户输入。这样我们就可以研究多种可行方法并对其优劣进行评估。我们还将实现软件状态机、练习使用中断并学习使用一些新的外围设备。

12.2　准备

除了 MPLAB IDE、MPLAB C32 编译器以及 MPLAB SIM 仿真器在内的这些常见软件工具之外，本章还需要用到 Explorer 16 演示板和你自行选择的在线调试器。你还需要在手边准备一个电烙铁和一些元器件，从而可以通过原型板区或者小扩展板来扩展 Explorer 16 演示板的功能。你还可以访问本书配套网站（www.exploringPIC32.com）来获取有关扩展板的更多信息，从而更好地完成本章中的实验。

12.3　按钮和机械开关

读取来自按钮和机械开关的输入是嵌入式控制应用中最常见的行为之一。但最终所有从端口引脚读取的输入信息还是会以数字信号来表示。单片机具备较高的运行速度，而开关又具备一些机械（弹性）特性，因此需要我们关注这个问题。

在图 12-1 中，4 个按钮中的一个会和 Explorer 16 演示板相连接。空闲状态下，开关保持开

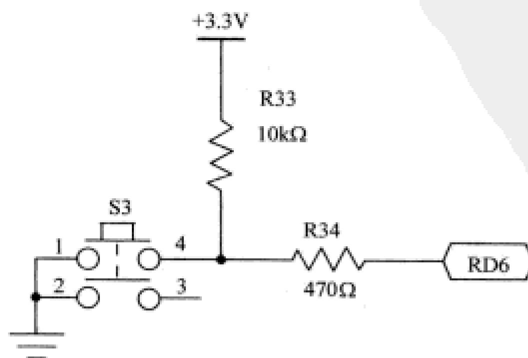


图 12-1　Explorer 16 演示板的按钮布局

路，输入引脚接上拉电阻保持逻辑高电平。按下按钮时，触头闭合，输入引脚变为逻辑低电平。如果把开关看成一个理想器件，那么两个状态之间的转换会立刻完成，毫不含糊，但事实上会有一点点不同。如图 12-2 所示，按下按钮并产生物理连接后，并不能得到一个利索的电平转换。所用材料的弹性特性、触头的表面氧化物以及其他一些因素都会导致电平转换变成一个跳变序列，随着设备的老化和磨损，序列中跳变的次数会增加，时间也会变长。这种现象通常称为触头的弹跳效应（contact bouncing），最坏情况下会持续几百微秒，甚至几毫秒。

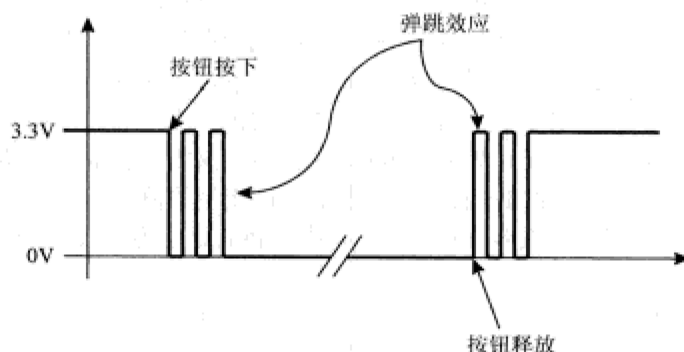


图 12-2　机械开关的电气响应

释放按钮时，两个触头表面之间的压力消失，电路断开，从而也会发生类似的弹跳效应。

对于运行在高时钟频率下的 PIC32 来说，这种弹跳事件的持续时间相对来说是非常大的。对输入信号线状态的密集轮询会检测到每一次弹跳，并将其计为对按钮的多次不同的按压和释放。因此，第一个实验中，我们将设计一小段代码来模拟此事件，从而获知 Explorer 16 演示板上按钮的“质量”。

创建一个新工程，名称为 Buttons，向其中加入一个新的源文件，名称为 bounce.c：

```
/*
** bounce.c
**
*/
// configuration bit settings, Fcy=72MHz, Fpb=36MHz
#pragma config POSCMOD=XT, FNOSC=PRIPLL
#pragma config FPLLIDIV=DIV_2, FPLLMUL=MUL_18, FPLLODIV=DIV_1
#pragma config FPBDIV=DIV_2, FWDTEN=OFF, CP=OFF, BWP=OFF
#include <p32xxxx.h>
main( void)
{
  int count;    // the bounces counter

  count = 0;
  // main loop
  while( 1)
  {
    // wait for the button to be pressed
    while (_RD6);

    // count one more button press
    count++;
```

```
    // wait for the button to be released
    while ( ! _RD6);

  } // main loop

} // main
```

初始化计数器之后，程序直接进入主循环，等待演示板上最左边的按钮（标示为 S3，并和 RD6 输入引脚相连接）被按下（转换为逻辑低电平）。一旦检测到按钮上的压力，就将计数器值加 1，并进入到下一个等待按钮释放的循环中，从而能够重新从头开始进行主循环。

马上生成该工程，并使用你自选的在线调试器在 Explorer 16 演示板上调试代码。为了完成这第一个实验，现在就运行该代码，并慢慢按下 S3 按钮多次，次数定为一个预设的数值：就算 20 次吧！停止程序的执行，查看变量 count 的当前值。仅需要将鼠标移动到编辑窗口中的该变量上，就可以看到一个小的弹出消息框，其中显示该变量的值（需要在 MPLAB 中启用该功能）；或者打开 Watch 窗口，将变量 count 加入其中（将其显示方式设为 Decimal，也就是十进制）。

在我个人的实验中，当我按下 20 次以后，获得的 count 值通常在 21 到 25 之间。正如汽车制造商所说："您的里程数可能不精确！"这其实是一个非常好的结果，表示大多数时候根本就没有发生弹跳。这个实验表明触头质量不错，但同时也表明演示板上的按钮到目前为止使用得还很少。如果准备设计会用到按钮和机械开关的应用程序，就应该考虑到最坏情况，在产品的有效使用期内其性能必将逐步下降。

12.4 封装按钮输入信号

设想一个封装方案，它可以应用到 Explorer 16 演示板上的这 4 个按钮上，并能在需要时扩展到一组更多的按钮上。我们从一个简单的函数开始设计，该函数收集所有的输入信号，并将其编码在单个整数中。将之前的源文件另存为 Buttons.c，并加入到工程中（替换 bounce.c）：

```
int readK( void)
{ // returns 0..F if keys pressed, 0 = none
  int c = 0;

  if ( !_RD6) // leftmost button
    c |=8;
  if ( !_RD7)
    c |=4;
  if ( !_RA7)
    c |=2;
  if ( !_RD13) // rightmost button
    c |=1;

  return c;
} // readK
```

Explorer 16 演示板的设计者将对应于这 4 个按钮的输入引脚分开放置于两个端口之间并不连续的区域上。设计者这样做可能是为了方便演示板布局，而并没有想到方便软件设计者的使用。

代码中所示的函数 readK() 接收 4 个输入信号，然后将其连续地封装于一个整数中，并将其作为函数的返回值。图 12-3 阐述了编码原理。



图 12-3 readK()按钮编码

因此，按钮的位置就由函数返回值的单个位的相对位置反映出来，其中最高有效位（位 3）对应于最左边的按钮的状态。同时，对每个输入信号的逻辑电平值进行了取反操作，从而用 1 来表示按下的按钮。因此，在空闲状态（即没有按钮被按下时）调用此函数，将返回 0。如果所有的按钮都被按下，返回 0x0f。

请注意目前还没有执行任何弹跳消除（debouncing）操作。所有的 readK()函数都可以看成输入信号状态的一幅图，将输入信号状态用数字这种更方便的格式来表示。如果有一个按钮阵列，以 3×4、4×4 或者更大的区域进行排列，对函数的修改也是很容易的，同时可以保持输出格式不变，也不需要改动接下来我们要编写的其他代码。

我们可以很快地修改 main()函数，使其采用我们在本书前面的内容中开发的 LCD.h 库函数，在 LCD 显示屏上显示输出结果。

```
main( void)
{
  char s[16];
  int b;

  initLCD();        // init LCD display

  // main loop
  while( 1)
  {
    clrLCD();
    putsLCD( "Press any button\n");
    b = readK();
    sprintf( s, "Code = %X", b);
    putsLCD( s);
    Delayms( 100);

  } // main loop
} // main
```

把 LCDlib.c 模块加入到工程源文件列表中，重新生成该工程，并使用在线调试器对 Explorer 16 演示板进行编程。

在运行这个简单的演示程序时，你可以发现，一旦按下某个按钮，就会立刻显示一个新的编码。同时按下多个按钮，则会产生一个位于 0x01 到 0x0f 之间的不固定的值。

为了方便起见，将 readK()函数加入到 explore.c 库模块中。如果你使用本书附带资源中提供的代码，就会注意到该函数已经存在，只是名称变为了 readKEY()，从而不会和示例代码产生冲突。

12.5 消除按钮输入弹跳

现在进行弹跳消除。弹跳消除其实就是把机械开关的错误电平转换全部进行过滤，其基本

技术就是在检测到第一次错误转换后加入一个短延时，延时之后再验证输出是否达到一个稳定的状态。而按钮释放后，则再加入一个短延时，并在延时之后验证输出是否处于空闲状态。

以下是新函数 getK()的代码，执行上面所说的 4 个步骤以及以下代码：

```
int getK( void)
{ // wait for a key pressed and debounce
  int i=0, r=0, j=0;
  int c;

  // 1. wait for a key pressed for at least .1sec
  do{
    Delayms( 10);
    if ( (c = readKEY()))
    {
      if ( c>r)              // if more than one button pressed
        r = c;               // take the new code
      i++;
    }
    else
      i=0;
} while ( i<10);
```

在 1 中，有一个 do…while 循环，每 10ms 循环一次，使用 readKEY()函数检测输入状态。该循环设计为在 10 个循环（总共 100ms）之内一直没有发生弹跳效应时才停止。在这段时间内，用户可能按下多个按钮。该函数适用于随着时间的推移，一个或多个按钮被依次按下的情况，而不是假设多个按钮以绝对同步的速度一起被按下。变量 r 中存放的代码值指明了在这段时间内被按下的所有按钮。

```
  // 2. wait for key released for at least .1 sec
  i =0;
  do {
    Delayms( 10);
    if ( (c = readKEY()))
    {
      if (c>r)    // if more then one button pressed
        r = c;    // take the new code
      i=0;
      j++;        // keep counting
    }
    else
      i++;
} while ( i<10);
```

在 2 中，情况跟 1 完全相反，目的是检测按钮的释放。do…while 循环设计用来等待所有的按钮被释放，直到输入稳定在空闲状态下，并保持至少 100ms。

```
// 3. check if a button was pushed longer than 500ms
if ( j>50)
  r+=0x80;       // add a flag in bit 7 of the code
```

在 3 中，实际上是利用了一个额外的计数器，用变量 j 来表示。它其实在第 2 个循环中就已经加入了，任务是检测按钮按下的持续时间是否超过了一个给定的阈值。此处设置为 500ms。当这种情况发生时，额外的标志位（位 7）会加到返回的编码中。这就很方便地给接口提供了额外的功能，而不需要在 Explorer 16 演示板上加入额外的硬件（按钮）。因此，比如说按下最左边的按钮一小段时间，就会产生编码 0x08。但是如果按下它超过半秒钟，就会返回编码 0x88。

```
  // 4. return code
  return r;
} // getK
```

在 4 中，把变量 r 表示的按钮编码返回给调用程序。

为了测试新功能并验证按钮弹跳效应已经消除，现在用以下代码替换 main() 函数，并保存为 Buttons2.c 文件：

```
main( void)
{
  char s[16];
  int b;

  initLCD();        // init LCD display
  putsLCD( "Press any button\n");

  // main loop
  while( 1)
  {
    b = getK();
    sprintf( s, "Code = %X", b);
    clrLCD();
    putsLCD( s);
  } // main loop
} // main
```

记住在工程中加入 lib 目录下的 LCDlib.c 和 explore.c 模块。

将工程源文件列表中的 buttons.c 文件用 Buttons2.c 替换，并重新生成该工程。用在线调试器对 Explorer 16 演示板进行编程之后，运行该代码并观察 LCD 显示屏上的结果。

首先可以注意到，结果和之前的示例程序运行结果刚好相反，新的代码仅在按钮被释放之后才在显示屏上有所显示。函数 getK() 实际上是一个*阻断函数*（blocking function）。它等待用户输入并仅在新的返回编码准备好以后才返回。

对多个按钮进行组合演示，同时按下 2 个或者 3 个按钮，并观察按下和释放的顺序是如何做到不会影响输出结果从而简化用户输入的。再试一下长按和短按的组合。你可以修改阈值，甚至加入新的阈值，检测超长时间的按钮按下动作。

由于该函数的作用重要，这里再一次推荐你把 getK() 函数加入到 explore.c 库模块中。如果使用本书附带资源中的代码，就会发现它已经更名为 getKEY()，从而避免和本章中的示例代码相冲突。

12.6 旋转编码器

另一种基于机械开关（有时候由光电传感器替代）的输入设备是*旋转编码器*（rotary encoder），在很多嵌入式控制应用中非常常见。在前面的内容中，我们也试过将电位器附加到 PIC32 的 ADC 模块上来提供用户输入（并控制吃豆人 Pac-Man 的位置）的用法，但是旋转编码器是纯粹的数字设备，具备很高的自由度，其主要优点是在任何旋转方向上都没有位移限制。有些编码器必须在它的*绝对位置*上才能提供信息，而另一些设计简单、成本低廉的编码器[被称为*增量式编码器*（incremental encoder）]则只能提供相对位移指示。

在嵌入式应用中，绝对式旋转编码器可用于识别电机轴或驱动轴的位置（角度）。增量式编码器可用于检测位移的方向以及电机的速度，也可在用户接口中作为在显示面板的菜单系统中选择项目的快速输入工具来使用，想象一下汽车导航仪和数字式收音机上无所不在的输入旋

钮吧。增量式编码器作为用户接口应用的另一个绝佳例子是（球形）鼠标（现在已经停产了）。它包含 2 个（光电）旋转编码器，检测两个方向上的相对位移。事实上，思考一下就会发现，计算机其实并不能及时知道鼠标究竟在什么位置，但是它知道你移动了多远，以及向什么方向移动。当然不要用现在的光电鼠标来做实验，它们的原理是完全不一样的。

为了对简单而便宜的旋转编码器（我使用 Bourns 的 ICW 型号）进行实验，此处建议你按图 12-4 所示，在 Explorer 16 演示板的原型板区焊接一对电阻（10kΩ）并在编码器和 PIC32 的 I/O 引脚间接上 3 根电线，这也算是对自己的原型设计能力的一次测验。

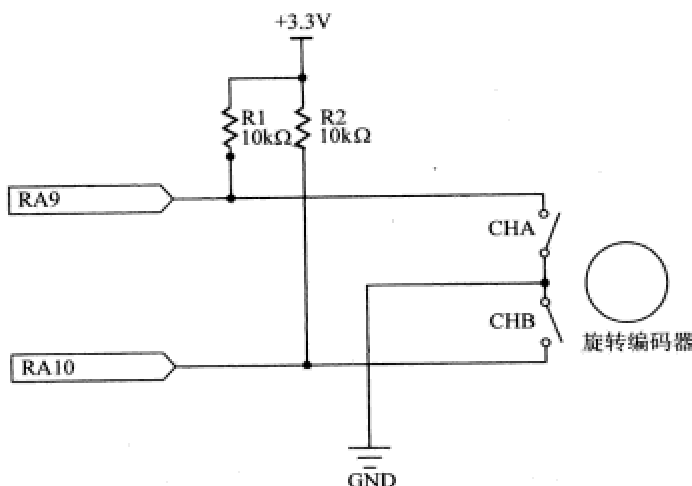


图 12-4　旋转编码图接口

接好以后，编码器就能提供两个很容易被 PIC32 所理解的输出波形（如图 12-5 所示）。注意编码器的位移是在制动器位置之间按步骤进行的。每一步编码器都生成两个换相过程，分别位于两个机械开关上，并对应于某个输入引脚。两个换相过程的顺序则告知了旋转的方向。因为两个波形除了有 90°的相位差之外，波形完全相同，所以这个简单的编码器通常也被称为正交编码器（quadrature encoder）。

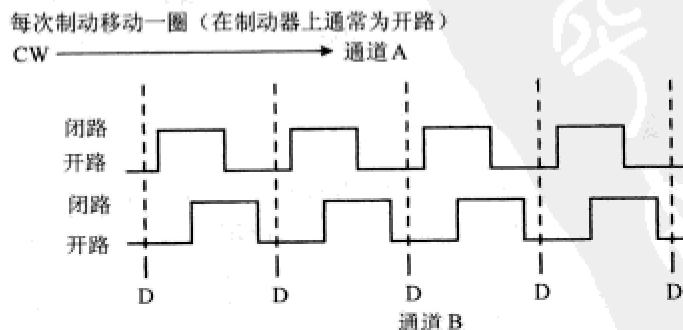


图 12-5　旋转编码器输出波形

在静止时，两个开关都是打开的，相应的输入引脚也上拉为逻辑高电平。顺时针旋转时，

CHA 开关先闭合，将 RA9 输入引脚电平变低，然后 CHB 开关闭合，将 RA10 输入引脚电平变低。而逆时针旋转时，顺序则刚好相反。当编码器到达下一个制动位置时，两个开关又同时变为打开状态。

以下是一个简单的程序，说明了如何连接旋转编码器，从而获取旋转按钮的位置，并在 LCD 显示屏上显示一个相对计数值。

```
/*
** Rotary.c
**
*/
// configuration bit settings, Fcy=72MHz, Fpb=36MHz
#pragma config POSCMOD=XT, FNOSC=PRIPLL
#pragma config FPLLIDIV=DIV_2, FPLLMUL=MUL_18, FPLLODIV=DIV_1
#pragma config FPBDIV=DIV_2, FWDTEN=OFF, CP=OFF, BWP=OFF

#include <p32xxxx.h>
#include <explore.h>
#include <LCD.h>

#define ENCHA  _RA9          // channel A
#define ENCHB  _RA10         // channel B

main( void)
{
  int i = 0;
  char s[16];

  initLCD();

  // main loop
  while( 1)
  {
    while( ENCHA);            // detect CHA falling edge
    Delayms( 5);              // debounce
    i += ENCHB ? 1 : -1;
    while( !ENCHA);           // wait for CHA rising edge
    Delayms( 5);              // debounce

    // display relative counter value
    clrLCD();
    sprintf( s, "%d", i);
    putsLCD( s);
  } // main loop

} // main
```

这样编写主循环中的代码的依据是一个简单的观察到的事实：通过观察某个输入的换相过程（比如说 ENCHA）可以检测到位移的发生。在该输入变化发生以后，立即观察另一个输入 ENCHB 的状态，则可以确定位移的方向。这可以从图 12-5 中看出来，你从左向右看（对应于顺时针旋转），当 CHA 开关闭合时（以上升沿表示），CHB 开关依然是打开的（低电平）。但如果从右向左看这幅图（对应于编码器的逆时针旋转），那么就会发现当 CHA 闭合时（上升沿），CHB 开关已经闭合了（高电平）。

我们刚学过开关弹跳消除方法，因此在代码中加入了 2 次延时例程的调用，用于保证在确实只有一次换相发生时，不会读到多次换相过程。延时的长度则取决于编码器生产厂商的设备数据手册上提供的信息。ICW 编码器的触头在转速为 15RPM 时，弹跳发生的时长最多为 5ms。

创建一个新工程，名为 Rotary。将这段代码保存为 rotary.c，并记住向工程源文件列表中加

入默认的 include 目录以及 lib 库中的 LCDlib.c 和 explore.c 源文件。

生成该工程，并对 Explorer 16 进行重新编程，然后运行该程序。

如果一切正常，你可以看到，在旋转编码器按钮时，LCD 显示屏上会连续地以十进制显示一个计数器的值。该计数器是一个 32 位的有符号整数，它会根据转动的方向以及转动的时间长度在不同大小的正值和负值之间摆动。

12.7　中断驱动的旋转编码器输入

刚才开发的这个简单的演示程序还存在一个主要问题，它必须假设单片机的所有部件全部用于执行一个任务，即检测 CHA 和 CHB 输入引脚的电平转换。在应用程序等待用户输入并且没有其他任务等待单片机执行时，这是可以接受的。但是，如果有比该任务优先级更高更重要的应用程序存在，而且这种情况又是经常发生的，那么就不能用"封闭式"输入算法这么奢侈的程序了，而是需要将编码器放入后台任务中。

从第 5 章的学习中得知，要想在嵌入式控制应用中获得多任务功能，最简单的办法就是采用 PIC32 的中断机制。后台程序变成了一个遵守一定规则的小型状态机。在我们的实验中，将之前所写的代码转换成状态机并画出其转化图（图 12-6），仅需要两个状态：

- 空闲状态（R_IDLE），当 CHA 编码器输入无效时；
- 有效状态（R_DETECT），当 CHA 编码器输入有效时。

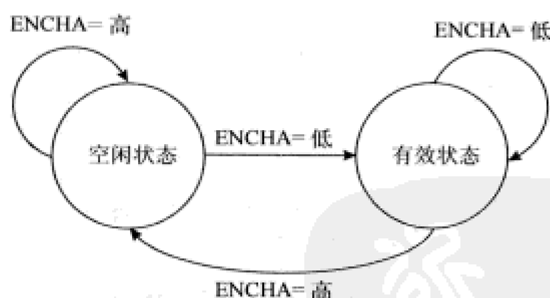


图 12-6　旋转编码器状态机框图

表 12-1 简单地阐述了两个状态之间的转换关系。

表 12-1　旋转编码器状态机状态转换

状　态	条　件	效　果
R_IDLE	ENCHA 有效（低电平）	如果 ENCHB 有效，旋转方向为逆时针（d=−1） 转换到 R_DETECT 状态
	ENCHA 无效（高电平）	设置默认旋转方向为顺时针（d=1） 保持当前状态不变（等待）
R_DETECT	ENCHA 无效（高电平）	更新计数器 转换到 R_IDLE 状态
	ENCHA 有效（低电平）	保持当前状态（等待）

将状态机的执行和定时器（比如 Timer2）产生的周期性中断绑定在一起，就可以保证只要时序合适，该程序会一直执行下去并在执行过程中自然地进行弹跳消除。

创建一个新的源文件，名称为 Rotary2.c，在文件头部加入常规的模板说明和几个变量声明：

```
/*
** Rotary2.c
**
*/
// configuration bit settings, Fcy=72MHz, Fpb=36MHz
#pragma config POSCMOD=XT, FNOSC=PRIPLL
#pragma config FPLLIDIV=DIV_2, FPLLMUL=MUL_18, FPLLODIV=DIV_1
#pragma config FPBDIV=DIV_2, FWDTEN=OFF, CP=OFF, BWP=OFF

#include <p32xxxx.h>
#include <plib.h>
#include <explore.h>
#include <LCD.h>

#define ENCHA _RA9         // encoder channel A
#define ENCHB _RA10        // encoder channel B
#define TPMS (FPB/1000)    // PB clock ticks per ms

// state machine definitions
#define R_IDLE      0
#define R_DETECT    1

volatile int RCount;
char RState;
```

注意变量 RCount 是用于维护相对位移的计数器，声明为 volatile 类型，从而告诉编译器该值会根据中断服务例程（即状态机）进行变化。因为该变量在主循环中是不会被写入的，这样就能保证编译器不会通过错误的假设而把代码优化成在 main() 函数中对其进行访问。

为了效率起见，选择 PIC32 的向量中断机制，书写中断服务例程如下：

```
void __ISR( _TIMER_2_VECTOR, ipl1) T2Interrupt( void)
{
  static char d;

  switch ( RState) {
    default:
    case R_IDLE:          // waiting for CHA rise
      if ( ! ENCHA)
      {
        RState = R_DETECT;
        if ( ! ENCHB)
          d = -1;
      }
      else
        d = 1;
      break;

    case R_DETECT:        // waitin for CHA fall
      if ( ENCHA)
      {
        RState = R_IDLE;
        RCount += d;
      }
      break;
  } // switch
```

```
  mT2ClearIntFlag();
} // T2 Interrupt
```

最后，需要写一个小的初始化例程来建立 Timer 2（5ms 一个周期）和状态机能够正确运行所需的初始条件：

```
void initR( void)
{
  // init state machine
  RCount = 0;           // init counter
  RState = 0;           // init state machine

  // init Timer2
  T2CON = 0x8020;       // enable Timer2, Fpb/4
  PR2 = 5*TPMS/4;       // 5ms period
  mT2SetIntPriority( 1);
  mT2ClearIntFlag();
  mT2IntEnable( 1);
} // init R
```

因为状态机的工作与绝对的时序无关，所以 Timer 2 的优先级可以设置为最低级 1 级。甚至在编码器旋转得非常快时（设备的数据手册表明最快速度为 120RPM），电平转换所需的时间也是处理器处理时间的几何级数倍（20ms）。应用程序中的其他任何任务都可以设置为更高一些的优先级。

最后，设计一个新的 main() 函数，将旋转编码器例程放入其中，周期性地（1s 10 次）检测 RCount 变量的值，并将其当前值显示在 LCD 屏幕上。

```
main( void)
{
  int i = 0;
  char s[16];

  initEX16();           // init and enable interrupts
  initLCD();            // init LCD module
  initR();              // init Rotary Encoder

  // main loop
  while( 1)
  {
    Delayms( 100);      // place holder for a complex app.

    clrLCD();
    sprintf( s, "RCount = %d", RCount);
    putsLCD( s);

  } // main loop

} // main
```

注意对 initEX16() 函数的调用。如果你还记得第 10 章的学习内容，就会注意除了仔细地对 PIC32 进行精细调试获取更好性能以外，还必须使能其向量中断模式。

同时注意在 main() 函数中对 Delayms(100) 调用的位置，你其实可以直接用其他复杂程序的核心代码替代它，它会持续运行而不会被编码器检测例程“阻挡”。

12.8 键盘

如果一组按钮、一个袖珍键盘或者一个旋转编码器可以为嵌入式控制应用提供价格低廉的用户输入方式，那么它们组合起来完全可以和真实的计算机键盘相媲美。

随着 USB 总线的出现，计算机终于可以从自打第一台 IBM PC 诞生以来的几十年中"遗传"下来的大量接口中解脱了。PS/2 鼠标和键盘接口即其中之一。这种变化带来的结果就是大量的"老"键盘充斥着二手市场，即使是全新的 PS/2 键盘，价格也很低廉。这给我们接下来要开发的 PIC32 工程创造了一个很好的机会，使其能获得强大的输入能力，复杂性不高，并且花钱还很少。



注解　将 PIC32 和 USB 键盘进行连接则是完全不同的方式。你需要用到一个 USB 主接口，这意味着其硬件和软件都很复杂。新的包含 USB 主接口的 PIC32 型号会着重考虑这些需求，但是其使用方法和 USB 协议所需的命令都超出了本书的讨论范围。

12.9 PS/2 物理接口

PS/2 接口采用一个 5 针的 DIN 或者 6 针的 mini-DIN 连接器（如图 12-7 所示）。第一种在最初的 IBM PC-XT 和 AT 系列中极为常见，但是并没有使用多久。更小的 6 针接口在最近几年才得以广泛应用。通过观察这两种不同接口的输出信号，你会发现它们的电气特性其实是一样的。

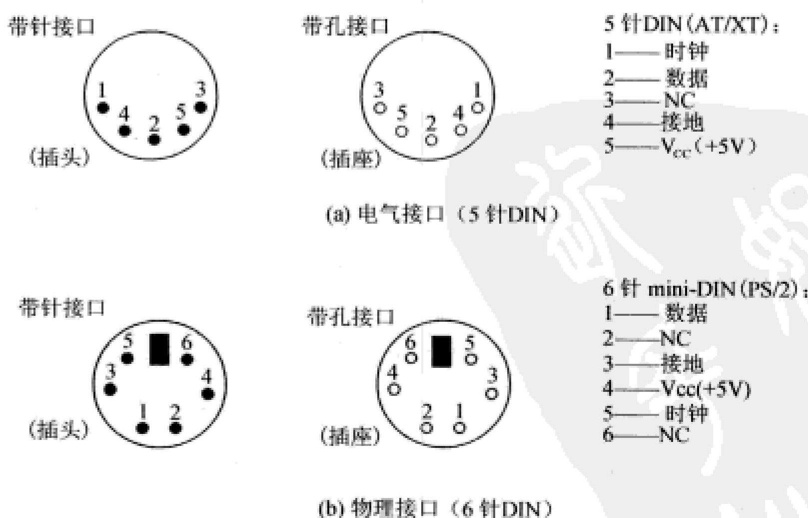


图 12-7　PS/2 接口连接器

主机必须提供 5V 的电压。电流大小则随着键盘型号和出厂时间的推移而不同，但是可以知道是在 50~100mA。（最早的规格中常常需要最高 275mA 的电流。）

数据线和时钟线都是带有上拉电阻（1~10kΩ）的开路连接器，可以进行两路通信。在正常操作模式下，由键盘对这两组线进行驱动，从而将数据送入计算机中。但是在需要时，计算机也可以取得控制权，对键盘进行配置，并改变 LED 灯的状态（CapsLock 按键和 NumLock 按键状态）。

12.10 PS/2 通信协议

在空闲状态下，数据线和时钟线都由上拉电阻（位于键盘内部）保持为高电平状态。在这种情况下，键盘使能，并且一旦某个按键按下，就能立即发送数据。如果主机使时钟线保持为低电平并持续 100μs 以上，后续的键盘传输都会暂停。如果主机将数据线电平也变低，然后恢复时钟线为高电平，就表示这是一个发送命令的请求（如图 12-8 所示）。

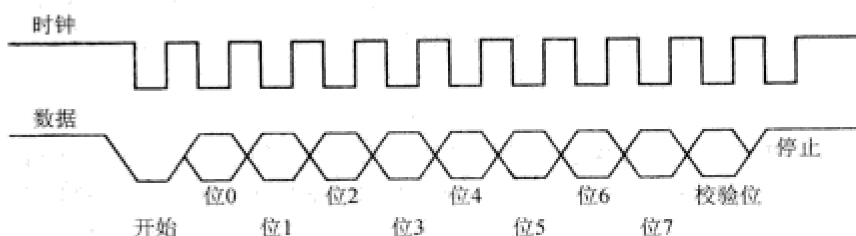


图 12-8 键盘到主机的通信波形

在本书前面的内容中已经讲过，PS/2 通信协议是一个同步通信协议和异步通信协议的奇妙组合。因为提供了时钟信号线，所以说它是同步的；但是因为使用开始信号、结束信号和奇偶位将 8 位数据包分隔开，所以它又像是一个异步协议。但键盘使用的波特率也并不是标称值，随着时间的推移、温度和月相的变化，它会一点一点地变化。典型的波特率值会在 10kbit/s 到 16kbit/s 间变化。数据在时钟信号为高电平时才会改变。时钟信号线为低电平时数据是有效的。不管数据是从主机传递到键盘还是从键盘传递到主机，总是由键盘来产生时钟信号。

> **注解** USB 总线转换了外围设备的角色，因为它使每个外围设备都成为主机的一个同步从设备。这大大简化了像 Windows 这样的非实时、非初级多任务操作系统的工作。串行端口和并行端口是类似的异步接口，并且因为 USB 总线出现的缘故，它们都过时了。

12.11 PIC32 和 PS/2 相连接

PS/2 通信协议的特性使 PS/2 键盘和 PIC32 的连接变成一个有趣的挑战，因为不管是 PIC32 的 SPI 接口，还是 UART 接口，都不能使用。实际上，SPI 接口不接收 11 位字（通常为 8 位字或者 16 位字），而 PIC32 的 UART 接口则需要周期性地传输特殊分隔字符，这样才能使用其强大的自动波特率检测功能。同样需要注意的是，PS/2 协议需要 5V 的电平信号，因此在选择直接与 PIC32 相连接的引脚时，要考虑其电平值。实际上，只有 5V 的数据输入引脚可以使用，因此这里面不包括和 ADC 输入多路器复用的 I/O 引脚。

12.12 输入捕获模块

第一个浮现在我脑海中的念头是，采用输入捕获模块（Input Capture Module）用软件实现一个 PS/2 串行接口外围设备。

在 PIC32MX360F512L 上，有 5 个可用的输入捕获模块，依次和 IC1~IC5 引脚相连接，这 5 个引脚又同时和 8、9、10、11 和 12 PORTD 引脚复用，见图 12-9。

每个输入捕获模块分别由单独的关联控制寄存器 ICxCON 进行控制，并与 Timer 2 或 Timer 3

组合起来，共同工作。

以下几种事件可以触发输入信号的捕获：

- ❑ 上升沿
- ❑ 下降沿
- ❑ 上升和下降沿
- ❑ 第 4 个上升沿
- ❑ 第 16 个下降沿

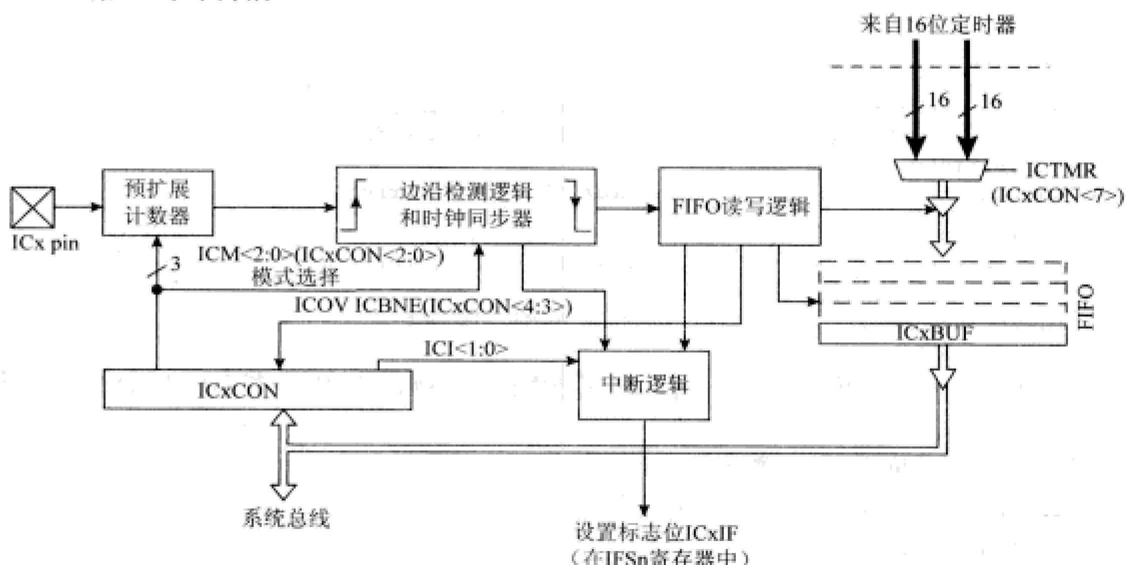


图 12-9 输入捕获模块框图

所选定时器的当前值记录并保存在 FIFO 缓冲区中，通过读取相应的 ICxBUF 寄存器可以获取该值。除了捕获事件以外，在一定数量的事件（每次、每秒、每三分之一秒或每四分之一秒）之后还可以产生一个中断，事件的数量是可编程的。

为了利用输入捕获模块接收来自 PS/2 键盘的数据流，可以将 IC1 的输入引脚（RD8）连接到时钟信号线，并将输入捕获模块配置成在每个时钟下降沿产生中断（如图 12-10 所示）。

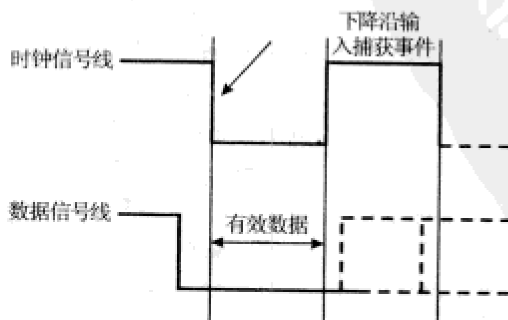


图 12-10 PS/2 接口位时序和输入捕获触发事件

创建一个新工程，名称为 IC，将新的 PS2IC.c 源文件加入到该工程中。在 PS2IC.c 中，将

以下初始化代码加入到通用模板说明之后：

```
#define PS2DAT  _RG12       // PS2 Data input pin
#define PS2CLK  _RD8        // PS2 Clock input pin (IC1)
void initKBD( void)
{
  // init I/Os
  _TRISD8 = 1;    // make RD8, IC1 an input pin, PS2 clock
  _TRISG12 = 1;   // make RG12 an input pin, PS2 data

  // clear the kbd flag
  KBDReady = 0;

  // init input capture
  IC1CON = 0x8082;            // TMR2, int every cap, fall'n edge
  mIC1ClearIntFlag();         // clear the interrupt flag
  mIC1SetIntPriority( 1);
  mIC1IntEnable( 1);          // enable the IC1 interrupt

  // init Timer2
  mT2ClearIntFlag();          // clear the timer interrupt flag
  mT2SetIntPriority( 1);
  mT2IntEnable( 1);           // enable (TMR2 is not active yet)
} // init KBD
```

同样需要为 IC1 中断向量创建一个中断服务例程。该例程必须作为状态机来运行，并按照以下步骤依次进行。

（1）验证开始位的存在（数据线变低）。

（2）将数据进行 8 位移位，计算校验位。

（3）验证有效校验位。

（4）验证停止位的存在（数据线变高）。

如果以上任何检测失败，状态机都会复位，返回到初始状态。接收到有效数据之后，会将其存放在缓冲区（试着把它想象成一个邮箱）中并设置一个标志位，从而让主程序或者任何其他“消费者”例程得知已经接收到了合法的按键码，并已做好接收的准备。为了获取有效按键码，必须先从邮箱中将其复制出来，并清除标志位（如图 12-11 所示）。

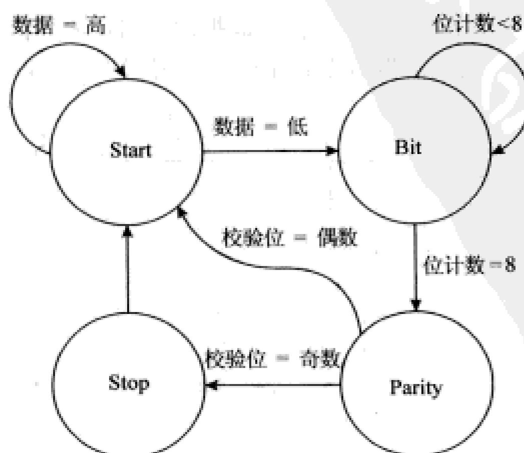


图 12-11　PS/2 接收状态机示意图

该状态机只需要 4 个状态和 1 个计数器。具体的状态转换情况如表 12-2 所示。

```
// definition of the keyboard PS/2 state machine
#define PS2START      0
#define PS2BIT        1
#define PS2PARITY     2
#define PS2STOP       3
#define TPS      (FPB/1000000)    // timer ticks per uS
#define TMAX     500*TPS          // 500uS time out limit

// PS2 KBD state machine and buffer
int PS2State;
unsigned char KBDBuf;
int KCount, KParity;

// mailbox
volatile int KBDReady;
volatile unsigned char KBDCode;
```

表 12-2 PS/2 接收状态机转换

状态	条件	效果
Start	数据=低	初始化位计数器 初始化奇偶校验位 转换到 Bit 状态
Bit	位计数<8	切换到按键码，最先接收最低位（向右移位） 更新校验位 递增位计数器
	位计数=8	转换到 Parity 状态
Parity	校验位=偶数	错误，转换到 Start 状态
	校验位=奇数	转换到 Stop 状态
Stop	数据=低	错误，转换到 Start 状态
	数据=高	将按键码保存到缓冲区中 设置标志位 转换到 Start 状态

理论上来说，如果将每次 Bit 状态进入到不同的位计数值都看成一个独立的状态，我认为这是一个 11 个状态的状态机。但是对于高效的 C 语言实现方式来说，4 个状态的模型工作起来是最好的方式。我们先定义一些用于维护状态机运行的常量和变量：

最后，输入捕获模块 IC1 的中断服务例程可以采用一个简单的 switch 语句来实现：

```
void __ISR( _INPUT_CAPTURE_1_VECTOR, ipl1) IC1Interrupt( void)
{ // input capture interrupt service routine
  int d;

  // 1. reset timer on every edge
  TMR2 = 0;
switch( PS2State){
default:
case PS2START:
  if ( ! PS2DAT)              // verify start bit
  {
    KCount = 8;               // init bit counter
    KParity = 0;              // init parity check
    PR2 = TMAX;               // init timer period
```

```
      T2CON = 0x8000;            // enable TMR2, 1:1
      PS2State = PS2BIT;
    }
    break;

  case PS2BIT:
    KBDBuf >>=1;                 // shift in data bit
    if ( PS2DAT)
      KBDBuf += 0x80;
    KParity ^= KBDBuf;           // update parity
    if ( --KCount == 0)          // if all bit read, move on
      PS2State = PS2PARITY;
    break;

  case PS2PARITY:
    if ( PS2DAT)                 // verify parity bit
      KParity ^= 0x80;
    if ( KParity & 0x80)         // if parity odd, continue
      PS2State = PS2STOP;
    else
      PS2State = PS2START;
    break;

  case PS2STOP:
    if ( PS2DAT)                 // verify stop bit
    {
      KBDCode = KBDBuf;          // save code in mail box
      KBDReady = 1;              // set flag, code available
      T2CON = 0;                 // stop the timer
    }
    PS2State = PS2START;
    break;

  } // switch state machine
   // clear interrupt flag
   d = IC1BUF;                    // discard capture
   mIC1ClearIntFlag();

 } // IC1 Interrupt
```

12.13　用激励脚本进行测试

可以利用小的打孔原型板区将 PS/2 mini-DIN 连接器和 Explorer 16 演示板连接起来，还有一个可选方案是为扩展连接器开发一个可定制的子板（PICTail）。不过，在决定设计这样的子板之前，必须保证选择的输出引脚和代码都是可工作的。MPLAB SIM 软件仿真器将再次成为我们选择的工具。

在前面的内容中，我们已经使用过软件仿真器，将它和 Watch 窗口、StopWatch 和 Logic Analyzer 组合起来，验证程序产生的时序和输出是否正确，但是这次还需要模拟输入。对于这一点，MPLAB SIM 提供了相当多的选项和资源，其数量之多以至于让仿真器看起来有点可怕。首先，仿真器提供了两种类型的输入激励：

- ❑ 异步激励，通常由用户手动触发；
- ❑ 同步激励，由仿真器在脚本给定的时间之后自动触发（用处理器周期或者秒数来表示）。

包含同步激励描述（有可能相当复杂）的脚本用 Stimulus 窗口来生成（如图 12-12 所示）。你必须将 MPLAB SIM 选择为主动调试工具（Debugger|Select Tool|MPLAB SIM），再从调试器

菜单中选择 Stimulus|New Workbook 打开 Stimulus 窗口。为了生成最简单的激励脚本类型，即在给定时间点给指定的输入引脚（同时也包括所有寄存器）赋值，用户可以选择第一个选项卡，Pin/Register Actions。

在选择测量单位以后，此处我们选择毫秒，单击表格第一行中占据对话框窗口的大部分空间的部位（会显示“Click here to Add Signals”），用户即可以向表中加入新列。为每个将要进行输入激励的引脚都加入一列。在我们的工程中，要加入新列的引脚是作为 PS/2 数据信号线的 RG12 引脚，以及连接到 PS/2 时钟信号线上的输入捕获模块的 IC1 引脚。此时可以开始编辑激励时序表了。为了模拟一次常见的 PS/2 键盘传输，需要产生一个 11 个周期的 10kHz 的时钟信号，和图 12-6 中给出的 PS/2 键盘波形一样。这需要在时序表中每隔 50μs 插入一个事件。作为示例，表 12-3 给出了我推荐的加入到 Stimulus 窗口时序表中的触发事件，用来模拟按键代码 0x79 的传输。

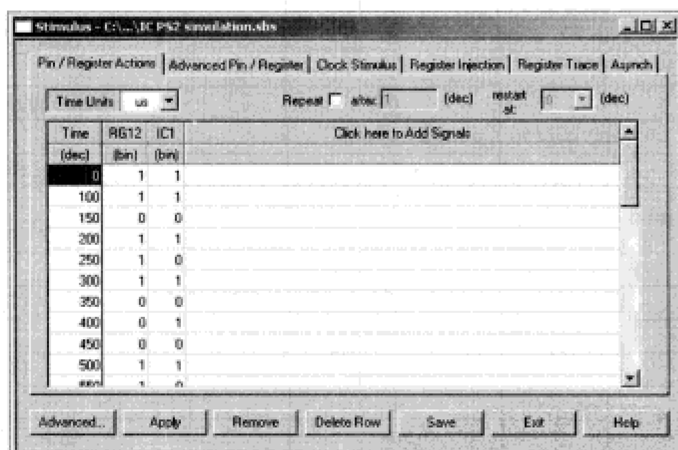


图 12-12 Stimulus 窗口

时序表填完以后，用户可以使用 Save 按钮将当前内容保存起来以备后用。生成的文件是后缀为.SBS 的 ASCII 文件。理论上可以利用 MPLAB IDE 的编辑器或者任何基本的 ASCII 编辑器手工修改该文件，但是强烈建议你不要这么做。该文件的格式要求非常严格，因此对其进行修改很有可能产生错误。如果你对看起来很简单的表格还使用“工作本”（workbook）这么“庞大”的名称来命名而感到疑惑，可以看一下 Stimulus 窗口中的其他面板（通过单击对话框顶部的选项卡来访问）；可以看到在本例中用到的方法只是众多可用方案中的一种。一个工作本文件可以包含大量的由任何一个或多个面板生成的不同类型的激励。

```
Segment of the Stimulus workbook file
## SCL Builder Setup File: Do not edit!!

## VERSION: 3.60.00.00
## FORMAT:  v2.00.01
## DEVICE:  PIC32MX360F512L

## PINREGACTIONS
us
No Repeat
RG12
IC1
```

```
--
0
1
1
--
100
1
1
--
150
0
0
```

表 12-3 基本的 SCL 时序发生器示例

时间（μs）	RG12	IC1	说　明
0	1	1	空闲状态，所有信号线上拉
100	1	1	
150	0	0	第一个下降沿，开始位（0）
200	1	1	
250	1	0	位 0，按键码的最低位（1）
300	0	1	
350	0	0	位 1（0）
400	0	1	
450	0	0	位 2（0）
500	1	1	
550	1	0	位 3（1）
600	1	1	
650	1	0	位 4（1）
700	1	1	
750	1	0	位 5（1）
800	1	1	
850	1	0	位 6（1）
900	0	1	
950	0	0	位 7，按键码的最高位（0）
1000	0	1	
1050	0	0	校验位（0）
1100	1	1	
1150	1	0	停止位（1）
1200	1	1	空闲状态

在开始使用生成的激励文件之前，我们必须再做一点工作来完成整个工程。首先编写一个头文件，定义可以访问的函数 initKBD()、标志位 KBDREADY 以及存放接收按键码 KBDCode 的缓冲区：

```
/*
**
** PS2IC.h
**
** PS/2 keyboard input library using input capture
*/
extern volatile int KBDReady;
extern volatile unsigned char KBDCode;

void initKBD( void);
```

需要注意的是，不要在 PS/2 接收方的实现代码中涉及内部工作的其他任何细节。这样可以保证在之后的工作中，不用修改接口就能采用多种不同的方法。把上述文件保存为 PS2IC.h 并加入到工程中。

同样创建一个新文件，名称为 PS2ICTest.c，它将包含通用模板说明、main() 函数例程，并使用 PS2IC.c 模块来测试其功能：

```
/*
** PS2ICTest.c
**
*/
 // configuration bit settings, Fcy=72MHz, Fpb=36MHz
 #pragma config POSCMOD=XT, FNOSC=PRIPLL
 #pragma config FPLLIDIV=DIV_2, FPLLMUL=MUL_18, FPLLODIV=DIV_1
 #pragma config FPBDIV=DIV_2, FWDTEN=OFF, CP=OFF, BWP=OFF

 #include <p32xxxx.h>
 #include <explore.h>
 #include "PS2IC.h"

 main()
 {
   int Key;
   initEX16();              // init and enable interrupts
   initKBD();               // initialization routine
   while ( 1)
   {
     if ( KBDReady)         // wait for the flag
     {
       Key = KBDCode;       // fetch the key code
       KBDReady = 0;        // clear the flag
     }
   } // main loop
 } //main
```

initEX16() 函数负责对 PIC32 进行精确调整来获取更好性能，同时使能向量中断模块。initKBD() 函数负责 PS/2 状态机的初始化，设置所选的输入引脚并配置输入捕获模块的中断。主循环等待中断例程设置 KBDready 标志位，这标志着按键代码已经准备好了，它将从缓冲区中获取按键代码并将它复制到本地变量 Key 中。最后，它清除 KBDReady 标志位，从而准备接收下一个新字符。

现在要记得将文件加入到工程中，并重新编译。在开始仿真之前，再次选择 Stimulus 窗口，并单击 Apply 按钮。



注解 保持 Stimulus 窗口处于打开状态（在后台）。不要单击 Exit 按钮，否则会关掉工作本并停止仿真。

单击 Reset 按钮（或者选择 Debugger|Reset）并观察在 0 微秒触发事件发生时第一个激励的到达。根据时间表格显示，RG12 和 IC1 信号线都应该设置为高。Output 窗口中的消息也证实了这一点（见图 12-13）。

现在是通过单步执行还是通过程序驱动来验证其执行的正确性，由你自己来决定。我的建议是在主循环中复制 KBDCode 到 Key 变量的语句处设置一个断点（breakpoint）。打开 Watch

窗口，将 Key 加入到标号列表中，然后单击 RUN 按钮。

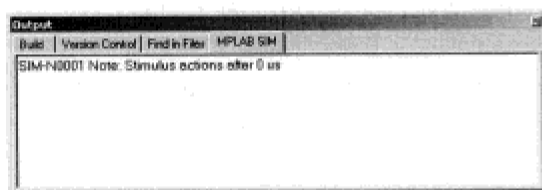


图 12-13　在 Output 窗口中（MPLAB SIM 面板），一个激励行为已经触发

几秒钟以后，执行将暂停在断点处，Key 的内容会反映出通过仿真的 PS/2 激励脚本发送的数据：0x79！

12.14　仿真器的运行特性统计工具

如果你想了解 PIC32 的仿真在 PC 机上的运行时间，那么可以利用 MPLAB SIM 的 Debugger 菜单中的一个有趣的选项：运行特性（profile）。选择 Profile 子菜单（Debugger|Profile）并单击 Reset Profile（如图 12-14 所示）。

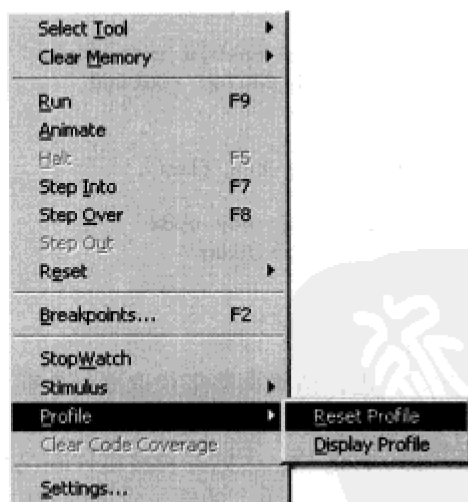


图 12-14　仿真器 Profile 子菜单

这将清除仿真器运行特性统计的计数器和定时器。单击 Reset 按钮并重复一次仿真（Debugger|Run）直到再次运行到断点处。这次选择 Debugger|Profiler|Display Profile 来显示来自 MPLAB SIM 的最新统计数据（如图 12-15 所示）。

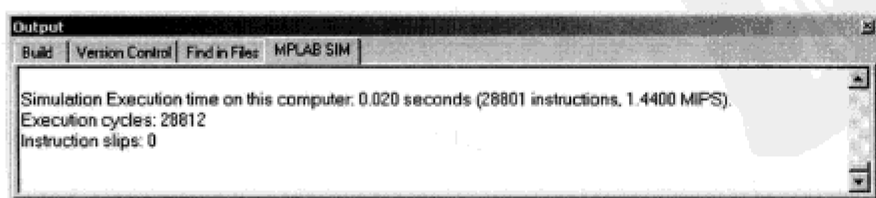


图 12-15　仿真器 Profile 输出

在输出窗口（MPLAB SIM 面板）中会给出一个相对长一些的报告，给出了在仿真过程中每条指令被处理器使用的次数，最底端给出了仿真速度的实际值的估值。在我运行的结果中，速度是 1.4MIPS。这个速度并不值得大书特书，但毕竟也是一个可靠的数据。其他 PIC 单片机仿真时，这些数值和实际的微处理器实时性能可能差不多，但是PIC32的仿真恰恰相反，和PIC32实际运行速度相比，软件仿真的速度（在我的电脑上）只是实际运行速度的 1/50！

12.15 变更通知模块

上一节中采用的输入捕获方法运行得非常好，但是我们仍然充满好奇心，想要探索一下其他和 PS/2 键盘进行有效连接的方式。特别是在 PIC32 上还有另一个有趣的外围设备，即变更通知（Change Notification，CN）模块，可以提供实现 PS/2 接口的方法。有多达 22 个 I/O 引脚和该模块相连，这也给了我们选择合适的 PS/2 接口输入引脚的自由，可以保证不和工程中其他功能相冲突，也不会和已经在 Explorer 16 演示板上使用的工程相冲突。

只有 3 个控制寄存器和 CN 模块相连。CNCON 寄存器包含使能该模块的基本控制位，CNEN 寄存器包含每个 CN 输入引脚的使能位。注意整个 CN 模块只有一个中断向量可用，因此在多个输入引脚使能的情况下，只能靠中断服务例程来决定究竟是哪个引脚状态发生了改变。最后一个寄存器 CNPUE 寄存器对每个输入引脚包含的内部上拉电阻的行为进行独立的控制（如图 12-16 所示）。

虚拟地址	名称		位 31/23/15/7	位 30/22/14/6	位 29/21/13/5	位 28/20/12/4	位 27/19/11/3	位 26/18/10/2	位 25/17/9/1	位 24/16/8/0
BF88_61C0	CNCON	31:24	—	—	—	—	—	—	—	—
		23:16	—	—	—	—	—	—	—	—
		15:8	ON	FRZ	SIDL	—	—	—	—	—
		7:0	—	—	—	—	—	—	—	—
BF88_61C4	CNCONCLR	31:0	对该寄存器执行写操作会清除 CNCON 中已选择的位，读操作未定义							
BF88_61C8	CNCONSET	31:0	对该寄存器执行写操作会设置 CNCON 中已选择的位，读操作未定义							
BF88_61CC	CNCONINV	31:0	对该寄存器执行写操作会对 CNCON 中已选择的位取反，读操作未定义							
BF88_61D0	CNEN	31:24	—	—	—	—	—	—	—	—
		23:16	—	—	CNEN 21[1]	CNEN 20[1]	CNEN 19[1]	CNEN 18	CNEN 17	CNEN 16
		15:8	CNEN[15:8]							
		7:0	CNEN[7:0]							
BF88_61D4	CNENCLR	31:0	对该寄存器执行写操作会清除 CNEN 中已选择的位，读操作未定义							
BF88_61D8	CNENSET	31:0	对该寄存器执行写操作会设置 CNEN 中已选择的位，读操作未定义							
BF88_61DC	CNENINV	31:0	对该寄存器执行写操作会对 CNEN 中已选择的位取反，读操作未定义							
BF88_61E0	CNPUE	31:24	—	—	—	—	—	—	—	—
		23:16	—	—	CNPUE 21[1]	CNPUE 20[1]	CNPUE 19[1]	CNPUE 18	CNPUE 17	CNPUE 16
		15:8	CNEN[15:8]							
		7:0	CNEN[7:0]							
BF88_61E4	CNPUECLR	31:0	对该寄存器执行写操作会清除 CNPUE 中已选择的位，读操作未定义							
BF88_61E8	CNPUESET	31:0	对该寄存器执行写操作会设置 CNPUE 中已选择的位，读操作未定义							
BF88_61EC	CNPUEINV	31:0	对该寄存器执行写操作会对 CNPUE 中已选择的位取反，读操作未定义							

注释 1：在 64 引脚的版本中，CNEN 和 CNPUE 的位未实现，全部读为 0

图 12-16 CN 控制寄存器表

在实际操作中，所有需要用来进行 PS/2 连接的只是一个 CN 模块输入引脚，用来和 PS/2 时钟信号线相连接。因为键盘本身已经提供了弱上拉电阻，所以在本例中并不需要 PIC32 的弱上拉功能。在 22 个可供选择的引脚中，我们将找到一个 CN 输入引脚，没有和 ADC 模块复用（同时需要承受 5V 的输入电压），同时也没有和 Explorer 16 演示板上的其他外围设备引脚重复。这需要对设备的数据手册和 Explorer 16 用户手册进行一点儿研究。但是一旦选定了输入引脚，比如 CN11（和引脚 RG9、SPI2 模块的 ss 信号线以及 PMP 模块的地址线 PMA2 复用），就可以写一个新的初始化例程了，这个初始化例程只需要几行代码（如图 12-17 所示）。

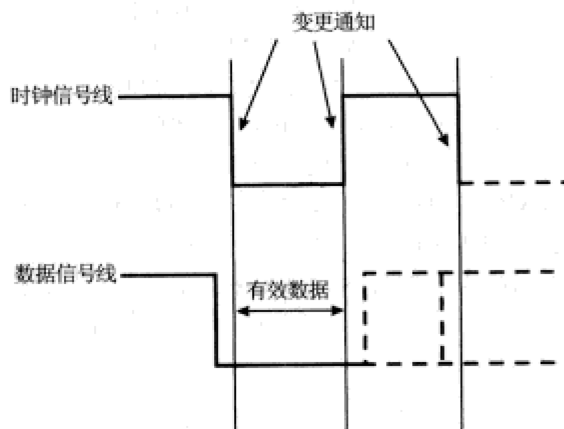


图 12-17 变更通知事件 PS/2 接口位时序

```
#define PS2DAT  _RG12     // PS2 Data input pin
#define PS2CLK  _RG9      // PS2 Clock input pin (CN11)

void initKBD( void)
{
   // init I/Os
   _TRISG9 = 1;            // make RG9 an input pin
   _TRISG12 = 1;           // make RG12 an input pin

   // clear the flag
   KBDReady = 0;

  // configure Change Notification system
  CNENbits.CNEN11 = 1;     // enable PS2CLK (CN11)
  CNCONbits.ON = 1;        // turn on Change Notification
  mCNSetIntPriority( 1);   // set interrupt priority >0
  mCNClearIntFlag();       // clear the interrupt flag
  mCNIntEnable( 1);        // enable interrupt
} // init KBD
```

根据中断服务例程的功能得知，可以使用和上一个示例中一模一样的状态机，同时再加入几行代码，从而保证中断服务是在时钟信号线的下降沿触发的。

实际上，使用输入捕获模块，只能选择在希望的时钟沿上产生中断，而使用 CN 模块则可以同时在上升沿和下降沿产生中断。在进入中断服务例程以后，立刻检查时钟信号线的状态，从而将两个时钟沿区分开来：

```
void __ISR( _CHANGE_NOTICE_VECTOR, ipl1) CNInterrupt( void)
{ // change notification interrupt service routine

  // 1. make sure it was a falling edge
  if ( PS2CLK == 0)
  {
    switch( PS2State){
    default:
    case PS2START:              // verify start bit
      if ( ! PS2DAT)
      {
        KCount = 8;             // init bit counter
        KParity = 0;            // init parity check
        PS2State = PS2BIT;
      }
      break;

    case PS2BIT:
      KBDBuf >>=1;              // shift in data bit
      if ( PS2DAT)
        KBDBuf += 0x80;
      KParity ^= KBDBuf;        // update parity
      if ( --KCount == 0)       // if all bit read, move on
        PS2State = PS2PARITY;
    break;
    case PS2PARITY:
      if ( PS2DAT)              // verify parity
        KParity ^= 0x80;
      if ( KParity & 0x80)      // if parity odd, continue
        PS2State = PS2STOP;
      else
        PS2State = PS2START;
      break;

    case PS2STOP:
      if ( PS2DAT)              // verify stop bit
      {
        KBDCode = KBDBuf;       // save code in mail box
        KBDReady = 1;           // set flag, code available
      }
      PS2State = PS2START;
      break;

    } // switch state machine
  } // if falling edge

  // clear interrupt flag
  mCNClearIntFlag();
} // CN Interrupt
```

将在前一个例子中用过的常量和变量声明加入到代码中：

```
// definition of the keyboard PS/2 state machine
#define PS2START      0
#define PS2BIT        1
#define PS2PARITY     2
#define PS2STOP       3

// PS2 KBD state machine and buffer
int PS2State;
unsigned char KBDBuf;
```

```
int KCount, KParity;

// mailbox
volatile int KBDReady;
volatile unsigned char KBDCode;
```

将以上代码打包成一个文件，将这个文件命名为 PS2CN.c。

头文件 PS2CN.h 和前一个例子一模一样，因为所需提供的是同一个接口：

```
/*
**
** PS2CN.h
**
** PS/2 keyboard input module using Change Notification
*/

extern volatile int KBDReady;
extern volatile unsigned char KBDCode;

void initKBD( void);
```

创建一个新的工程，名称为 PS2CN，将.c 文件和.h 文件都加入到工程中。

最后，创建一个主模块来测试本方法。这个主模块还是和上一个例子几乎完全相同：

```
/*
** PS2CNTest.c
**
*/
// configuration bit settings, Fcy=72MHz, Fpb=36MHz
#pragma config POSCMOD=XT, FNOSC=PRIPLL
#pragma config FPLLIDIV=DIV_2, FPLLMUL=MUL_18, FPLLODIV=DIV_1
#pragma config FPBDIV=DIV_2, FWDTEN=OFF, CP=OFF, BWP=OFF

#include <p32xxxx.h>
#include <explore.h>
#include "PS2CN.h"

main()
{
  initEX16();                   // init and enable interrupts
  initKBD();                    // kbd initialization

  while ( 1)
  {
    if ( KBDReady)              // wait for the flag
    {
      PORTA = KBDCode;   // fetch the key code
      KBDReady = 0;      // clear the flag
    }
  } // main loop
} //main
```

保存并重新生成工程（Project|BuildAll），将所有的模块编译并链接。为了测试变更通知方法，我们再一次使用 MPLAB SIM 的激励生成功能。重复在上一个工程中的大部分工作。从 Stimulus 窗口开始（Debugger|Stimulus|New Workbook），创建一个新的工作本。在窗口中创建两列，一列用于连接到 RG12 的 PS/2 数据线，但是这次 PS/2 的时钟信号线是和 CN 模块的 CN11 输入引脚相连接。将表 12-3 中的激励按顺序加入，把 IC1 input 列替换成 CN11 列。把工作本保存为 PS2CN.sbs，然后单击 Apply 按钮来激活激励脚本。

现在可以执行代码来测试新的 PS/2 接口功能的正确性。打开 Watch 窗口并把 Key 变量加入到符号表中。然后在主循环的 KBDCode 复制到 Key 变量处加入一个断点。最后，执行一个复位操作（Debugger|Reset）并验证第一个事件的触发（在 0μs 时将 PS/2 的两个输入线都置为高）。运行代码（Debugger|RUN），如果一切正常，就可以看到处理器运行不超过 1s 就在断点处停下来，Key 变量的内容已经更新，变成了按键码 0x79。我们又成功了！

12.16 开销评估

从输入捕获模块改变为变更通知模块的方法太容易了。这两个外围设备都非常强大，并且尽管设计目的不同，但是当应用到同一个任务时，它们的执行方式几乎相同。然而，在嵌入式领域中，即使可用的资源很多，用户仍然会经常考虑是否还能利用更少的资源来解决问题。本章中的几个示例也是如此。

通过计算这两种方法使用的资源，并对各自的相对不足之处进行比较，我们来衡量一下它们的实际开销。在使用输入捕获模块时，实际上只用上了 PIC32MX360F512L 上的 5 个 IC 模块中的一个。这个模块的设计目的是和定时器（Timer 2 或 Timer 3）配合使用，而在我们的程序中，并没有用到时序信息，而仅仅用到了和输入沿触发相关的中断机制。使用变更通知模块时，则仅仅使用了 22 个可选输入引脚中的一个，但是也同时控制了该设备所提供的唯一的中断向量。换句话说，如果需要用到变更通知模块的任何其他输入引脚，那就不得不和这些引脚共享中断向量，这必将带来额外的延时并增加了复杂性。因此我认为这两种方案基本上打了个平手。

12.17 I/O 轮询

还有一种方法可以用来和 PS/2 键盘相连接。这是最基本的一种方法，只需使用一个定时器来产生周期性的中断，再加上单片机的任何一个可接收 5V 电压的 I/O 引脚即可。从某种程度上来说，从配置和布局的角度来看，这是最灵活的一种方法。同时，这也是最通用的一种方法，因为任何型号的单片机，即使是最小和最便宜的，也会提供至少一个符合需求的定时器模块。这种方法所基于的理论也是非常简单的，即在规则的时间间隔内产生一个中断，时间间隔由和所选定时器相关联的周期寄存器的值来决定（如图 12-18 所示）。

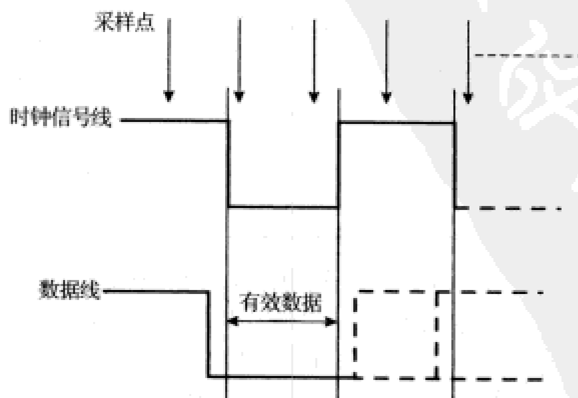


图 12-18 I/O 轮询方法采样点处的 PS/2 接口位时序

因为之前从没用过 Timer 4 以及跟它相关联的周期寄存器 PR4，所以在这次实验中练习一下使用 Timer 4。中断服务例程 T4Interrupt() 将对 PS/2 时钟信号线的状态进行采样，从而

确定在之前的一个周期内是否出现过下降沿。如果检测到下降沿，PS/2 数据线状态将被判定为接收按键码。为了确定执行采样的频率，从而确定最优化的 PR4 寄存器值，需要对 PS/2 时钟信号线上两个下降沿之间能允许的最短时间做一番研究。这由 PS/2 接口设定的最大位速率来决定。根据文档，该值大约为 16kbit/s。在这种速率下，时钟信号必须是方波形式，并且有大约 50%的占空比，周期大约为 62.5μs。换句话说，每次有数据位出现在 PS/2 的数据线上时，时钟信号线必须保持大于 30μs 的低电平，并且在下一个数据位出现之前，时钟信号线又必须保持差不多同样时长的高电平。

将 PR4 的值设置成能将中断周期控制在小于 30μs 之内（比如 25μs），就可以保证在时钟信号线的两个连续的沿之间，总是能至少被采样一次。但是，键盘传输位速率可能低至 10kbit/s，从而让两个沿之间的时间距离的最大值达到 50μs。在这种情况下，在两个时钟沿之间，时钟信号线和数据信号线有可能被采样 2 次，甚至多达 3 次。因此，必须建立一个新的状态机来检测真实的下降沿的发生，并保持对 PS/2 时钟信号的正确跟踪（如图 12-19 所示）。

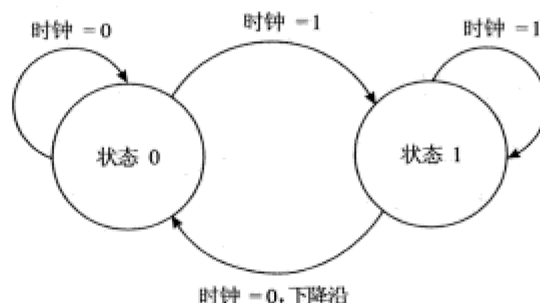


图 12-19　时钟轮询状态机图

状态机仅需要 2 个状态，所有的状态转换见表 12-4。

表 12-4　时钟轮询状态机转换

状　态	条　件	效　果
状态 0	时钟=0	停留在状态 0
	时钟=1	上升沿，转换到状态 1
状态 1	时钟=1	停留在状态 1
	时钟=0	检测到下降沿 执行数据状态机 转换到状态 0

检测到下降沿以后，仍然使用上一个工程中开发的状态机来读取数据信号线。需要注意的是，这时候并不能保证在时钟信号线上实际的下降沿发生之后，数据信号线上的值能被正确地采样，而是很有可能已经延迟。为了避免在有效时间段之外读取到错误的数据信号，就必须同步采样时钟信号线和数据信号线。在中断服务例程一开始就将两个输入值复制到两个本地变量中（d 和 k）即可。在本示例中，选择再次使用 RG12 作为数据线，RG13 作为时钟信号线。以下是之前阐述的时钟轮询状态机的实现框架：

```
#define PS2CLK  _RG13       // PS2 Clock output
#define PS2DAT  _RG12       // PS2 Data input pin

// PS2 KBD state machine and buffer
int PS2State;
unsigned char KBDBuf;

// mailbox
volatile int KBDReady;
volatile unsigned char KBDCode;

void __ISR( _TIMER_4_VECTOR, ipl1) T4Interrupt( void)
{
  int d, k;

  // sample the inputs clock and data at the same time
  d = PS2DAT;
  k = PS2CLK;
  // keyboard state machine
  if ( KState)
  { // previous time clock was high KState 1
    if ( !k)                // PS2CLK == 0
    { // falling edge detected,
      KState = 0;           // transition to State0

              <<<< insert data state machine here >>>>

    } // falling edge
    else
    { // clock still high, remain in State1

    } // clock still high
  } // state 1

  else
  { // state 0
    if ( k) // PS2CLK == 1
    { // rising edge, transition to State1
      KState = 1;

    } // rising edge
    else
    { // clocl still low, remain in State0

    } // clock still low
  } // state 0

  // clear the interrupt flag
  mT4ClearIntFlag();
} // T4 Interrupt
```

利用轮询机制的周期性特征，可以向 PS/2 接口加入一个新的特性，从而只用很小的开销就使其健壮性得到提高。首先，加入一个计数器到时钟状态机的两个状态的空闲循环中。这样的话，如果在传输过程中 PS/2 键盘被断开，或者因为任何原因导致接收例程的同步性被破坏，都能够通过超时判断来检测和修正错误状况。

新的状态转换表（见表 12-5）更新为包含超时计数器 KTimer。

表 12-5 时钟轮询（带超时判断）状态机转换表

状 态	条 件	效 果
状态 0	时钟=0	停留在状态 0 将 KTimer 递减 如果 KTimer=0，错误 复位数据状态机
	时钟=1	上升沿，转换到状态 1
状态 1	时钟=1	停留在状态 1 将 KTimer 递减 如果 KTimer=0，错误 复位数据状态机
	时钟=0	检测到下降沿 执行数据状态机 转换到状态 0 重启 KTimer

新的状态转换表只需要向中断服务例程中加入几条语句即可：

```
void __ISR( _TIMER_4_VECTOR, ipl1) T4Interrupt( void)
{
  int d, k;

  // sample the inputs clock and data at the same time
  d = PS2DAT;
  k = PS2CLK;

  // keyboard state machine
  if ( KState)
  { // previous time clock was high KState 1
    if ( !k)                          // PS2CLK = 0
    { // falling edge detected,
      KState = 0;                     // transition to State0
      KTimer = KMAX;                  // restart the counter

             <<<< insert data state machine here >>>>

    } // falling edge
    else
    { // clock still high, remain in State1

      KTimer--;
      if ( KTimer == 0)               // Timeout
        PS2State = PS2START;          // Reset data SM
    } // clock still high
  } // Kstate 1
  else
  { // Kstate 0
    if ( k)                           // PS2CLK == 1
    { // rising edge, transition to State1
      KState = 1;
    } // rising edge
    else
```

```
    { // clocl still low, remain in State0
      KTimer--;
      if ( KTimer = 0)                      // Timeout
        PS2State = PS2START;                // Reset data SM
    } // clock still low
  } // Kstate 0

  // clear the interrupt flag
  mT4ClearIntFlag();
} // T4 Interrupt
```

12.18 测试 I/O 轮询方法

现在对前一个工程中的数据状态机进行修改，使其对 d 和 k 中的值进行操作，d 和 k 在中断服务例程入口处就已采样并保存。新的数据状态机完全可以用一个 switch 语句来实现：

```
switch( PS2State){
  default:
  case PS2START:
    if ( !d) // PS2DAT == 0
    {
      KCount = 8;                       // init bit counter
      KParity = 0;                      // init parity check
      PS2State = PS2BIT;
    }
    break;
case PS2BIT:
  KBDBuf >>=1;            // shift in data bit
  if ( d)                 // PS2DAT == 1
    KBDBuf += 0x80;
  KParity ^= KBDBuf;      // calculate parity
  if ( --KCount == 0)     // all bit read
    PS2State = PS2PARITY;
  break;

case PS2PARITY:
  if ( d)                 // PS2DAT == 1
    KParity ^= 0x80;
  if ( KParity & 0x80)    // parity odd, continue
    PS2State = PS2STOP;
  else
    PS2State = PS2START;
  break;

case PS2STOP:
  if ( d)                 // PS2DAT == 1
  {
    KBDCode = KBDBuf;     // write in the buffer
    KBDReady = 1;
  }
  PS2State = PS2START;
  break;

} // switch
```

第 3 个模块是初始化例程：

```
void initKBD( void)
{
  // init I/Os
  ODCGbits.ODCG13 = 1;         // make RG13 open drain (PS2clk)
  _TRISG13 = 1;                // make RG13 an input pin (for now)
  _TRISG12 = 1;                // make RG12 an input pin

  // clear the kbd flag
  KBDReady = 0;
  // configure Timer4
  PR4 = 25*TPS - 1;         // 25 us
  T4CON = 0x8000;           // T4 on, prescaler 1:1
  mT4SetIntPriority( 1);    // lower priority
  mT4ClearIntFlag();        // clear interrupt flag
  mT4IntEnable( 1);         // enable interrupt
} // init KBD
```

这很直接。

将以上所有代码保存在名称为 PS2T4.c 的文件中，并创建一个新的头文件。

```
/*
**
** PS2T4.h
**
** PS/2 keyboard input library using T4 polling
*/

extern volatile int KBDReady;
extern volatile unsigned char KBDCode;

void initKBD( void);
```

这个头文件其实和前面所有模块的头文件是完全相同的，主测试模块也没有很大的不同：

```
/*
** PS2T4 Test
**
*/
// configuration bit settings, Fcy=72MHz, Fpb=36MHz
#pragma config POSCMOD=XT, FNOSC=PRIPLL
#pragma config FPLLIDIV=DIV_2, FPLLMUL=MUL_18, FPLLODIV=DIV_1
#pragma config FPBDIV=DIV_2, FWDTEN=OFF, CP=OFF, BWP=OFF

#include <p32xxxx.h>
#include <explore.h>
#include "PS2T4.h"

main()
{
   initEX16();                // init and configure interrupts
   initKBD();                 // initialization routine

  while ( 1)
  {
    if ( KBDReady)            // wait for the flag
    {
      PORTA = KBDCode;        // fetch the key code
      KBDReady = 0;           // clear the flag
    }
  } // main loop
} //main
```

创建一个新工程 T4，将以上 3 个文件全部加进去。对整个工程进行编译，并按照前面两个例子中同样的步骤生成激励脚本。记住这次时钟信号线上的激励必须提供在 RG13 引脚上。打开 Watch 窗口，将 PORTA 和 KBDCode 加入列表中。最后，在 PORTA 赋值处加入一个断点，并执行 Debug|Run。如果运行顺利，这次可以在 Watch 窗口中看到 PORTA 值的更新，并给出新的值 0x79！再次成功！

12.19 开销和效能的考虑

和前两种方法的开销相比，I/O 轮询方法在选择输入引脚上的自由度最高，并且仅使用一项资源（即定时器）和一个中断向量。如果有其他任务的定时周期可以折合为轮询周期的倍数，那么也可以和这些任务完全共享周期性中断，形成共同的定时平台。超时特性是一个额外的收获；如果想要在之前的方法中获得该功能，除了输入捕获模块或变更通知模块及中断以外，还必须再使用一个单独的定时器和另一个中断服务例程。

再看一下效能。输入捕获和变更通知方法看起来更高效，因为这两种方法仅在检测到时钟沿时才产生中断。实际上也正是如此，从这一点上来看输入捕获方法确实是最好的，因为采用该方法可以精确选择一种所需要的时钟沿类型，即 PS/2 时钟信号线的下降沿。

I/O 轮询方法看起来需要更长的中断服务例程，但是代码长短并不代表中断服务例程的实际分量。实际上，如果我们研究得更仔细些，看看形成 I/O 轮询中断服务例程的两个嵌套状态机就会发现，在每次调用时只有几条指令被执行，因此执行时间非常短，而开销也达到最小。

可以对实现 PS/2 接口的 3 种方法执行一个简单的测试，来验证中断服务例程带来的实际软件开销。在此仅用最后一种方法作为示例。分配一个 I/O 引脚（最好是选择一个 PORTA 中 JTAG 端口不使用的 LED 输出），用于形象化地观察微处理器什么时候正在处理中断服务例程。在中断服务例程入口处对该引脚置位，而在退出之前对其复位：

```
void __ISR(..) T4Interrupt( void)
{
  _RA2 = 1;        // flag up, inside the ISR

  <<< Interrupt service routine here >>

  _RA2 = 0;        // flag down, back to the main
}
```

使用 MPLAB SIM 仿真器的逻辑分析器窗口，在电脑屏幕上观察结果。根据 Logic Analyzer（逻辑分析仪）检查表可知，必须使能 Trace 缓冲区，并设置正确的仿真速度。选择 RA0 通道并重新生成整个工程。

为了测试前 2 种方法（IC 和 CN），必须打开 Stimulus 窗口并应用 scripts 来模拟输入信号。如果不这样，就根本不会有中断产生。在测试 I/O 轮询方法时，则并不需要这么做，Timer 4 会持续产生中断，而我们需要测试的也正是在没有键盘输入时，连续轮询所浪费的时间是多少。

让 MPLAB SIM 运行几秒钟，然后停止仿真，并切换回 Logic Analyzer 窗口。这个时候需要将视图比例放大一些，从而看得更加精确（如图 12-20 所示）。

单击 cursors 按钮并拖动它们来计算 RA2 信号线上两次连续的上升沿之间的时钟周期数，两次连续的上升沿也正表示两次连续地进入中断服务例程。因为选择了 25μs 的周期，因此两个调用之间的周期数应该是 900（25μs×36 周期/μs@72MHz）。

对 RA2 上升沿和下降沿之间的时钟周期数的测试则说明了花费在中断服务例程内部处理

上的时间；我测试的结果是 36 个时钟周期。这两个值的比值说明了 PS/2 接口所耗费的计算能力在整个设备计算能力之中所占比例。在我的测试中，这个值仅为 4%。

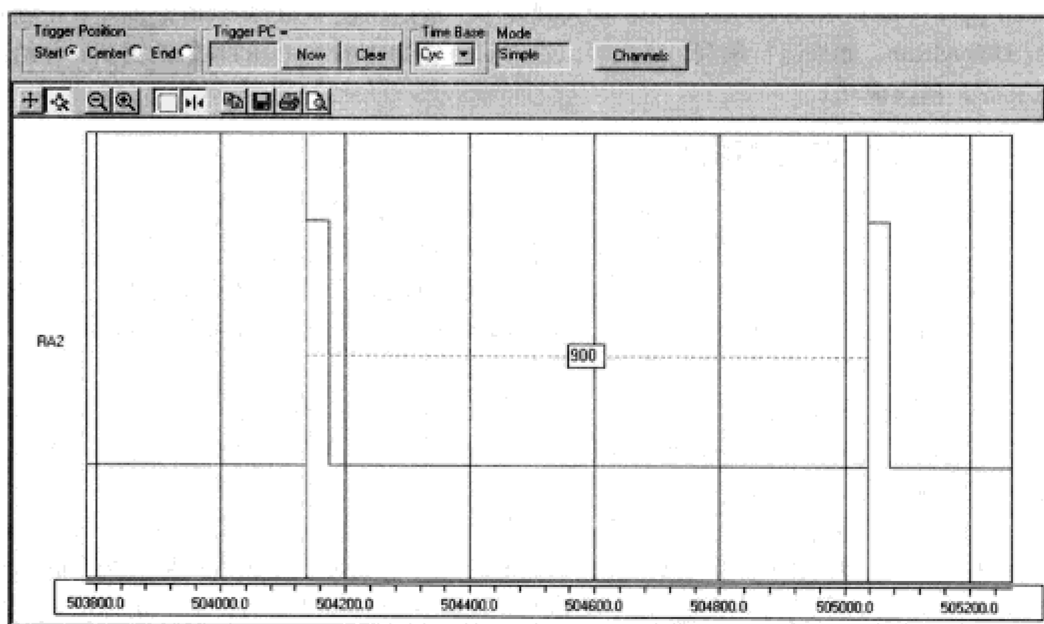


图 12-20　Logic Analyzer 窗口，计算 I/O 轮询方法的时钟周期数

12.20　键盘缓冲

我们还必须考虑一些独立于这 3 种连接方法之外的细节问题，才能说彻底完成了 PS/2 键盘的连接。首先，需要在 PS/2 接口例程和“消费者”程序或者说是主程序之间加入一个缓冲机制。目前我们只是提供了一个简单的邮箱机制，因此只能存储收到的最后一个按键码。如果进一步对 PS/2 键盘协议的工作机制进行研究，就会发现单个按键被按下并弹起时，最少有 3 个按键码（最多 5 个）传送到主机。如果考虑到 Shift、Ctrl 和 Alt 按键的组合，情况就会变得更复杂一些，单字节的邮箱显然不能满足需求。我的建议是使用至少包含 16 字节的先进先出（FIFO）缓冲区。向缓冲区输入数据的操作可以很容易地集成到接收方的中断服务例程中，从而在接收到新按键码时能将它立即放入 FIFO 缓冲区中。

缓冲区可以声明为一个字符串数组，两个指针以循环的方式分别指向缓冲区的头和尾（如图 12-21 所示）。

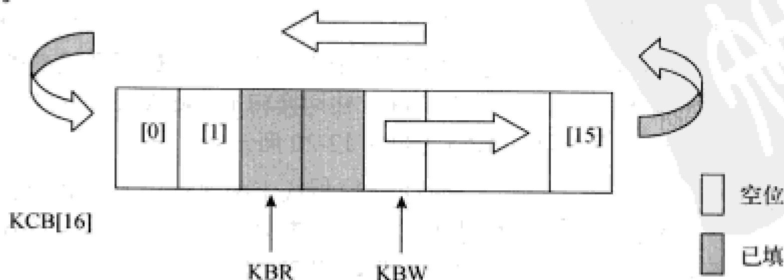


图 12-21　循环缓冲区（FIFO）

```
// circular buffer
unsigned char KCB[ KB_SIZE];

// head and tail or write and read pointers
volatile int KBR, KBW;
```

遵守以下几条简单的规则，就可以掌握缓冲区内容。

- 写指针 KBW（或者说是头指针）指向第一个空位，用于接收下一个按键码。
- 读指针 KBR（或者说是尾指针）指向第一个已填充的位置。
- 当缓冲区全空时，KBR 和 KBW 指向同一个位置。
- 当缓冲区填满时，KBW 指针指向 KBR 之前的位置。
- 向缓冲区写入一个字符或者从缓冲区读取一个字符之后，相应的指针将递增。
- 一旦到达数组末尾，两个指针都会回到数组的第一个位置。

把以下代码插入初始化例程中：

```
// init the circular buffer pointers

  KBR = 0;
  KBW = 0;
```

然后更新状态机的 STOP 状态：

```
case PS2STOP:
  if ( PS2IN & DATMASK)       // verify stop bit
  {
    KCB[ KBW] = KBDBuf;       // write in the buffer
    // check if buffer full
    if ( (KBW+1)%KB_SIZE != KBR)
      KBW++;                  // else increment ptr
    KBW %= KB_SIZE;           // wrap around
  }
  PS2State = PS2START;
  break;
```

操作符%的使用是为了获取用 KBW 指针当前位置除以缓冲区大小之后的余数。这样就能保证该指针在到达循环缓冲区末尾之后，能够再回到缓冲区头部。

从 FIFO 缓冲区读取按键码时，还需要注意几个问题。在选择输入捕获或变更通知方法时，需要编写一个新函数（getKeyCode()）来替代邮箱/标志位机制。如果缓冲区中没有按键码，就返回 FALSE；如果至少存在一个按键码，就返回 TRUE；按键码则通过指针返回：

```
int getKeyCode( char *c)
{
  if ( KBR == KBW)            // buffer empty
    return FALSE;

  // else buffer contains at least one key code
  *c = KCB[ KBR++];           // extract the first key code
  KBR %= KB_SIZE;             // wrap around the pointer

  return TRUE;
} // getKeyCode
```

getKeyCode()函数只是修改读指针；因此在中断使能时执行该操作是安全的。如果在读按键码的过程中发生了中断，那么可能发生以下两种情况。

- 缓冲区为空，新的按键码将被加入，但是 getKeyCode()函数只能在下一次调用时才能得知有新的字节填入。

- ❑ 缓冲区不为空，如果空间足够，中断例程将把新的字符加入到缓冲区尾部。

这两种情况都不会引起冲突或者错误结果。

但是，如果选择 I/O 轮询方法，定时器中断是一直进行着的，因此可以使用它再执行一个任务。这个任务的主要思想是维护一个简单的邮箱和标志位机制，用于将接口处的按键码传送给接收例程，并让中断服务例程不断地对邮箱进行检查，随时准备向其中填入来自 FIFO 的内容。这样一来，就可以将整个 FIFO 的管理完全交给中断服务例程来负责，而让缓冲区完全透明，邮箱传递接口也因此变得简单。针对 I/O 轮询机制的新的完整中断服务例程如下所示：

```
void __ISR( _TIMER_4_VECTOR, ipl1) T4Interrupt( void)
{
  int d, k;

//_RA2 =1;

  // 1. check if buffer available
  if ( !KBDReady && ( KBR!=KBW))
  {
    KBDCode = KCB[ KBR++];
    KBR %= KB_SIZE;
    KBDReady = 1;              // flag code available
  }

  // 2. sample the inputs clock and data at the same time
  d = PS2DAT;
  k = PS2CLK;

  // 3. Keyboard state machine
  if ( KState)
  { // previous time clock was high KState 1
    if ( !k)                   // PS2CLK == 0
    { // falling edge detected,
      KState = 0;              // transition to State0
      KTimer = KMAX;           // restart the counter

  switch( PS2State){
  default:
  case PS2START:
    if ( !d)// PS2DAT == 0
    {
      KCount = 8;             // init bit counter
      KParity = 0;            // init parity check
      PS2State = PS2BIT;
    }
    break;

  case PS2BIT:
    KBDBuf >>= 1;             // shift in data bit
    if ( d)                   // PS2DAT == 1
      KBDBuf += 0x80;
    KParity ^= KBDBuf;        // calculate parity
    if ( --KCount == 0)       // all bit read
      PS2State = PS2PARITY;
    break;

  case PS2PARITY:
    if ( d)                   // PS2DAT == 1
```

```
      KParity ^= 0x80;
    if ( KParity & 0x80)    // parity odd, continue
      PS2State = PS2STOP;
  else
      PS2State = PS2START;
  break;

  case PS2STOP:
    if ( d)                 // PS2DAT == 1
    {
      KCB[ KBW] = KBDBuf;  // write in the buffer
      // check if buffer full
      if ( (KBW+1)%KB_SIZE != KBR)
        KBW++;              // else increment ptr
      KBW %= KB_SIZE;       // wrap around
    }
    PS2State = PS2START;
    break;
        } // switch
      } // falling edge
      else
      { // clock still high, remain in State1
        KTimer--;
        if ( KTimer == 0)       // timeout
          PS2State = PS2START;  // reset data SM
      } // clock still high
    } // Kstate 1
    else
    { // Kstate 0
      if ( k)                   // PS2CLK == 1
      { // rising edge, transition to State1
        KState = 1;
      } // rising edge
      else
      { // clocl still low, remain in State0
        KTimer--;
        if ( KTimer == 0)       // timeout
          PS2State = PS2START;  // reset data SM
      } // clock still low
    } // Kstate 0

    // 4. clear the interrupt flag
    mT4ClearIntFlag();

  //_RA2 = 0;
  } // T4 Interrupt
```

12.21 按键码的解码

到目前为止我们一直在对按键码进行讨论，你可能觉得按键码和每个按键的 ASCII 码是一致的，也就是说，如果按下键盘上的 A 按键，那么发送的就应该是相应的 ASCII 代码（0x41）。但是事情并不是这么简单。为了保持键盘布局的中立性，所有的 PC 机键盘都使用扫描码（scan code），给每个按键分配一个数值，这个值和第一台 IBM PC（即 circa 1980）最早实现的键盘扫描固件有关。根据特定的（国际通行的）键盘布局，从扫描码到实际的 ASCII 字符的转换会在更高一级来完成，如今是由 Windows 驱动程序来执行的。一些历史原因导致目前至少存在 3 种

不同的并且保持部分兼容的“扫描码字符集”。值得庆幸的是，默认状态下，所有的键盘都支持扫描码字符集#2，也是在接下来的讲述中要集中讨论的这一种。

每按下一个按键（任何按键，包括 Shift 和 Ctrl 键），和它相关联的扫描码就会送到主机；这个代码称为通码（make code）。一旦该键弹起，新的按键码（序列）就会送到主机；这些代码称为断码（break code）。断码通常由相同的生成码加上 0xF0 前缀组成。有些按键的通码长度达到 2 字节（通常是 Ctrl、Alt 和方向键），那么断码也就会长达 3 字节（如表 12-6 所示）。

表 12-6 扫描码字符集#2 中使用的通码和断码示例（默认值）

按　键	通　码	断　码
A	1C	F0，1C
5	2E	F0，2E
F10	09	F0，09
右方向键	E0，74	E0，F0，74
右 Ctrl 键	E0，14	E0，F0，14

为了处理这些信息，并将扫描码转换成正确的 ASCII 码，必须需要一个对照表将基本的扫描码映射到给定的键盘布局上。以下代码给出了通用美式英语键盘布局的转换表：

```
// PS2 keyboard codes (standard set #2)
const char keyCodes[128]={
        0,   F9,    0,   F5,    F3,  F1,    F2,  F12,    //00
        0,  F10,   F8,   F6,    F4, TAB,   '`',    0,    //08
        0,    0,L_SHFT,  0,L_CTRL, 'q',   '1',    0,    //10
        0,    0,  'z',  's',   'a', 'w',   '2',    0,    //18
        0,  'c',  'x',  'd',   'e', '4',   '3',    0,    //20
        0,  ' ',  'v',  'f',   't', 'r',   '5',    0,    //28
        0,  'n',  'b',  'h',   'g', 'y',   '6',    0,    //30
        0,    0,  'm',  'j',   'u', '7',   '8',    0,    //38
        0,  ',',  'k',  'i',   'o', '0',   '9',    0,    //40
        0,  '.',  '/',  'l',   ';', 'p',   '-',    0,    //48
        0,    0, '\'',    0,   '[', '=',     0,    0,    //50
CAPS, R_SHFT,ENTER,  ']',    0,0x5c,    0,    0,        //58
    0,     0,    0,    0,    0,    0, BKSP,    0,        //60
    0,   '1',    0,  '4',  '7',    0,    0,    0,        //68
    0,   '.',  '2',  '5',  '6',  '8',  ESC,  NUM,        //70
  F11,   '+',  '3',  '-',  '*',  '9',    0,    0         //78
};
```

这个数组声明为常量，因此它将被分配到固定的程序内存空间里，并保存在宝贵的 RAM 中。同样，还需要一个类似的表，用于映射每个按键加上 Shift 功能以后的值：

```
const char keySCodes[128] = {
        0,    F9,     0,    F5,   F3,   F1,    F2,  F12,      //00
        0,   F10,    F8,    F6,   F4,  TAB,   '~',    0,      //08
        0,     0,L_SHFT,     0,L_CTRL, 'Q',  '!',    0,      //10
        0,     0,   'Z',   'S',  'A',  'W',   '@',    0,      //18
        0,   'C',   'X',   'D',  'E',  '$',   '#',    0,      //20
        0,   ' ',   'V',   'F',  'T',  'R',   '%',    0,      //28
        0,   'N',   'B',   'H',  'G',  'Y',   '^',    0,      //30
```

```
    0,    0,  'M',  'J',  'U', '&', '*',   0,       //38
    0,  '<',  'K',  'I',  'O', ')','('',   0,       //40
    0,  '>',  '?',  'L',  ':', 'P', '_',   0,       //48
    0,    0, '\"',    0,  '{', '+',   0,   0,       //50
 CAPS, R_SHFT,ENTER, '}',   0, '|',   0,   0,       //58
    0,    0,    0,    0,    0,   0,BKSP,   0,       //60
    0,  '1',    0,  '4',  '7',   0,   0,   0,       //68
    0,  '.',  '2',  '5',  '6', '8',ESC, NUM,        //70
  F11,  '+',  '3',  '-',  '*', '9',   0,   0        //78
 };
```

对于所有的 ASCII 字符，转换都是直截了当的，但是必须给功能按键、Shift 以及 Ctrl 按键分配特殊值。仅有少数特殊按键可以在 ASCII 字符集中找到对应的代码：

```
// special function characters
#define TAB     0x9
#define BKSP    0x8
#define ENTER   0xd
#define ESC     0x1b
```

所有其他特殊按键必须自定义映射码，只是在使用到这些按键时，将自定义的值忽略，然后赋值为通用码（0）：

```
#define L_SHFT   0x12
#define R_SHFT   0x12
#define CAPS     0x58
#define L_CTRL   0x0
#define NUM      0x0
#define F1       0x0
#define F2       0x0
#define F3       0x0
#define F4       0x0
#define F5       0x0
#define F6       0x0
#define F7       0x0
#define F8       0x0
#define F9       0x0
#define F10      0x0
#define F11      0x0
#define F12      0x0
```

getC() 函数将对大部分常用按键执行基本的转换，同时保持对 Shift 按键状态以及 Caps 按键状态的跟踪：

```
int CapsFlag=0;
char getC( void)
{
  unsigned char c;

  while( 1)
  {

    while( !KBDReady);// wait for a key to be pressed
    // check if it is a break code
    while (KBDCode == 0xf0)
    { // consume the break code
      KBDReady = 0;
      // wait for a new key code
```

```
        while ( !KBDReady);
        // check if the shift button is released
        if ( KBDCode == L_SHFT)
          CapsFlag = 0;
        // and discard it
        KBDReady = 0;
        // wait for the next key
        while ( !KBDReady);
      }
      // check for special keys
      if ( KBDCode == L_SHFT)
      {
        CapsFlag = 1;
        KBDReady = 0;
      }
      else if ( KBDCode == CAPS)
      {
        CapsFlag = !CapsFlag;
        KBDReady = 0;
      }

      else // translate into an ASCII code
      {
        if ( CapsFlag)
          c = keySCodes[KBDCode%128];
        else
          c = keyCodes[KBDCode%128];
        break;
      }

    }
    // consume the current character
    KBDReady = 0;
    return ( c);
} // getC
```

12.22　小结

本章对嵌入式控制应用中获取用户输入的几种常用方法进行了探索和研究。从最基本的按钮和机械开关的弹跳效应开始，我们研究了旋转编码器，同时分析了（PS/2）计算机键盘的连接难度。这也给了我们很好的练习使用两种新的外围设备模块的机会：输入捕获模块和变更通知模块。我们还讨论了实现 FIFO 循环缓冲区的方法，并着重研究了中断管理机制。同时也设法对 MPLAB SIM 仿真器有了新的了解，第一次使用它的异步输入激励测试了本章代码。全章都在重点研究如何在资源的使用和每种接口方法提供的性能之间达到平衡。

12.23　对 PIC24 行家的提示

PIC32 的 IC 模块和 PIC24 中的 IC 模块基本完全一样，不过仍然在设计中加入了一些重要的改进。在把 PIC24 程序移植到 PIC32 上时，以下所列几项主要不同之处会对代码产生影响。

（1）ICxCON 寄存器依据标准的外围设备模块布局，提供一个 ON 控制位，允许用户在不使用该模块时将其禁止从而降低功耗。

（2）在模块和定时器对（组成一个 32 位定时器）组合使用时，ICxC32 控制位允许 32 位

数据的捕获。

(3) IC 模块在新的模式 6 (ICxM=110) 下运行时，ICxFEDGE 控制位允许选择第一个时钟沿（上升沿或下降沿）。

PIC32 的 CN 模块和 PIC24 中的 CN 模块也基本完全一样，在设计中也进行了一些重要的改进。在把 PIC24 程序移植到 PIC32 上时，以下所列几项主要不同之处会对代码产生影响。

(1) 加入了一个新的 CNCON 寄存器，提供了一套标准的控制位，包括 ON、FRZ 和 IDL 来更好地管理低功耗模式下的模块行为。

(2) CNEN (32 位) 控制寄存器将 PIC24 中分布在两个单独的 (16 位) 寄存器 (CNEN1 和 CNEN2) 中的输入引脚使能位组合在了一起。

(3) CNPUE (32 位) 控制寄存器和 CNEN 相似，将 PIC24 中分布在两个单独的 (16 位) 寄存器 (CNPUE1 和 CNPUE2) 中的所有上拉使能位组合在了一起。

12.24 提示与技巧

每个 PS/2 键盘都有一个内部 FIFO 缓冲区，容纳 16 个按键码。那么在主机没有准备好接收时，键盘就具备了暂存用户输入的能力。在本章开始就提到过，在任何给定的时间点，主机都可以通过将时钟信号线拉低（至少持续 100μs）来暂停和键盘的通信，并且可以在某一时间段内使其一直保持低电平状态。当时钟信号线重新变高之后，键盘传输继续。如果最后一个按键码的传输被中断，它将重新传输，同时会将 FIFO 缓冲区中的内容全部传输到主机。

为了模拟主机暂停键盘传输的功能，我们必须使用开漏驱动产生一个输出来控制时钟信号线。幸运的是，PIC32 很容易做到这一点，因为它的 I/O 端口模块是可配置的。每个 I/O 端口有一个关联的控制寄存器 (ODCx)，可以对每一个引脚输出驱动进行单独配置，并使其工作在开漏模式下。

需要注意的是，将 PIC32 的输出和任何 5V 设备相连接时，这个功能都特别有用。在我们的示例中，将 PS/2 时钟信号线转换成开漏输出只需要几行代码：

```
_ODG13 = 1;    // cfg PORTG pin 13 output in open-drain mode
_LATG13 = 1;   // initially let the output in pull up
_TRISG13 = 0;  // enable the output driver
```

注意，和所有的 PIC 单片机一样，哪怕某个引脚被配置为输出信号，它的当前状态仍然能作为输入信号来读取。因此，可以一边传输命令，一边从键盘接收字符，而不必让引脚反复在输出和输入模式间切换。

12.25 练习

(1) 加入一个函数，发送命令到键盘来控制 LED 的状态并设置按键重复速率。

(2) 替换 stdio.h 库文件中的输入辅助函数_mon_getc()，将键盘输入重定向到 stdin 流输入中。

(3) 加入对 PS/2 鼠标接口的支持。

12.26 参考书

Ed Nisley 所著的 *The Embedded PCs ISA Bus*。该书讲述了几乎二十年来每个 IBM PC 机的遗留接口以及其核心 ISA 总线。如今这些东西仍然被用于一些工业控制设备（比如 PC104 平台）和嵌入式应用设备中。

12.27 链接

www.computer-engineering.org。这是一个很好的网站，可以找到很多关于PS/2键盘和鼠标接口的有用文档。

www.pc104.com/whatis.html。PC104平台，把IBM PC架构首次引入单片机的试验品之一，使其用于嵌入式控制。

第 13 章　视频处理

13.1　计划

最近出现了先进的玻璃基片技术（chip-on-glass，COG），LCD 显示屏在手机和很多消费类电子设备中也得到了广泛使用，这说明带有集成控制器的小型显示设备已经变得越来越普通和便宜了。集成控制器带有图像缓冲功能，并能执行简单的文本和图形命令，从而把应用程序从维护显示功能的繁重任务中解脱出来。但是，当我们想完全控制屏幕并产生动画效果时该怎么办呢？或者说想突破集成控制器的某些限制时该怎么办呢？

在本章，我们将介绍直接和电视屏幕或者说任何可以接收标准复合视频信号的显示设备相连接的技术。这是一个测试 PIC32 外围设备模块的新功能、学习新的编程技术的好机会。我们第一个工程的目的就是得到一块很漂亮的黑屏（完美同步的视频帧），不过很快就会用几个很有用又有趣的图形程序将它填充起来。

13.2　准备

除了 MPLAB IDE、MPLAB C32 编译器和 MPLAB SIM 模拟器在内的常用软件工具之外，本章还需要用到 Explorer 16 演示板和用户自行选择的在线调试器。同时需要一个电烙铁和一些新的元件，从而可以利用原型区或者小型扩展板来扩展演示板的功能。同时，你还可以访问本书配套网站（www.exploringPIC32.com）来获取有关扩展板的更多信息，从而更好地完成实验。

13.3　探索

在当今的视频研究领域中，有很多不同的格式和标准，但是只有最古老最常见的那一种，才被称为“复合”视频格式。商业市场上最早出现的电视机所采用的就是这种格式。现如今，它代表着所有视频显示方式（不管是最新型的现代化的高清平板电视机、DVD 播放机还是 VHS 磁带录像机）都具有的共同特征。所有的视频设备都基于同样的基本概念，即图像都是“画”出来的，每次画一条线，从屏幕的最左上角开始，水平地移动到右上角，然后快速地回到左边稍低一点的位置画第二条线，一直持续走这样的之字形路线，直到整个屏幕被画完。然后，重复这个过程，就能刷新整个图像。这个过程很快，以至于我们的肉眼被蒙骗，以为整个图像是同时显示出来的。如果是动画图像的话，同样也会以为是流动和连续的（如图 13-1 所示）。

在世界各地，每年都会有并不完全兼容的系统被开发出来，但是它们的基本机制都一直保持不变。改变的只是组成图像的扫描线数目、刷新频率以及色彩信息的编码方式。

表 13-1 给出了 3 种在美国、欧洲以及亚洲最常用的视频标准。这 3 种标准都将同步信息和“亮度”信息（即底层的黑白图像）编码在相似的复合信号中。图 13-2 给出了 NTSC 复合信号的详细信息。

所谓“复合”，是为了说明视频信号是将 3 种不同信息整合在一起来传输的。这 3 种信息包括实际的亮度信号以及水平和垂直同步信息。

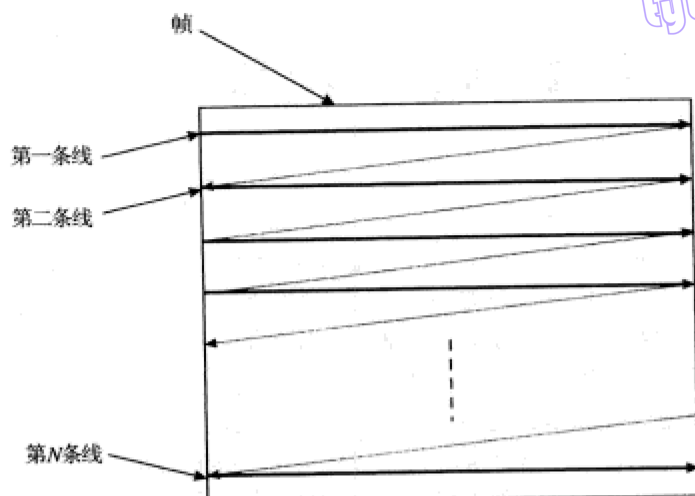


图 13-1 视频图像扫描

表 13-1 国际视频标准示例

项目 \ 地区	美 国	欧洲和亚洲	法国和其他地区
标准	NTSC	PAL	SECAM
每秒扫描的视频帧数目	29.97*	25	25
扫描线数量	525	625	625

*NTSC 每秒扫描 30 帧，但是新的颜色标准中将其变为了 29.97，从而能适应“色彩副载波”石英振荡器的特定频率。

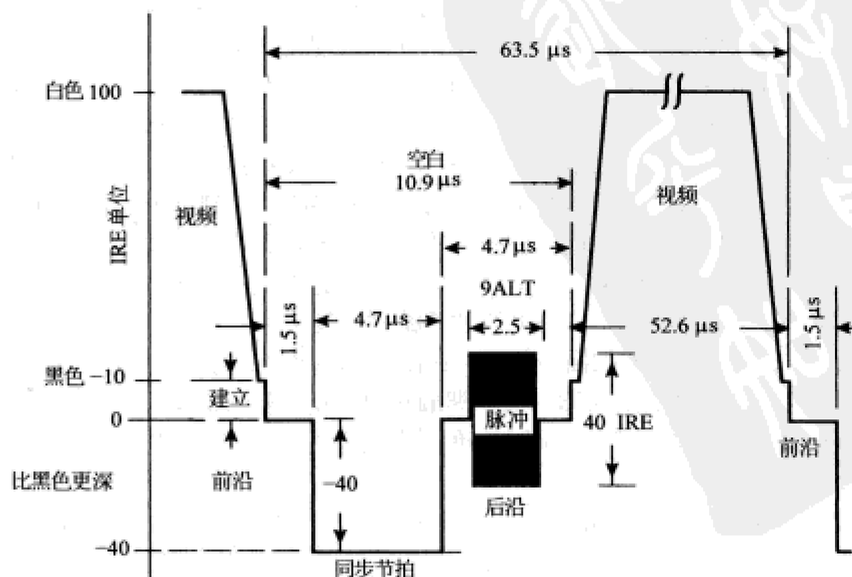


图 13-2 NTSC 复合信号水平扫描线详细信息

水平扫描线信号包括以下内容。

- 水平同步脉冲，由显示设备用于识别每条扫描线的开始。
- 所谓的“后沿”，图像周围黑色边框的最左边缘。
- 扫描线的实际亮度信息；电压越高，点的亮度越高。
- 所谓的“前沿”，产生图像的最右边缘。

色彩信息是单独传输的，用高频副载波来调制。后沿中部的一组突发短脉冲用于和副载波同步。3 种主要标准在色彩信息编码方法上有很大的不同，但是，如果仅仅只需要黑白图像的显示，就可以忽略大部分不同之处，并去掉色彩副载波同步脉冲。

这 3 项标准都采用了同一项技术，叫做隔行扫描（interlacing），能以较小的带宽提供（相对）较高分辨率的输出。在实际应用中，每个图像帧进行显示时，传输并画在屏幕上的并不是全部扫描线，而只有半数扫描线。偶数扫描线帧和奇数扫描线帧交替出现，整个图像看起来是以标准中设定的频率进行更新的（分别为 25Hz 和 30Hz），但实际的帧扫描频率其实是该频率的 2 倍。对于典型的电视信号广播，这种方法是相当有效的，但是在显示文本时，特别是显示水平扫描线时，会产生干扰闪烁。计算机显示器上经常会出现这种情况。

基于这个原因，所有的现代计算机显示器不再使用隔行扫描技术，而使用逐行扫描（progressive scanning）技术。大部分的现代电视机，尤其是那些采用液晶屏和等离子技术的电视机，还会对接收到的电视图像执行去隔行（deinterlacing）操作。我们在将要开发的工程中也不使用隔行扫描，但是为了得到更稳定、更具可读性的显示输出，还是要牺牲掉一半的图像分辨率。换句话说，我们将以 60 帧/秒的双倍速率来传输 262 扫描线的帧（NTSC 标准）。经常接触 PAL 或 SECAM 电视机/显示器的读者将会发现，如果想把工程修改为针对刷新率为 50 帧/秒的 312 扫描线的帧，相对来说也是很容易的。一个完整的视频帧信号如图 13-3 所示。

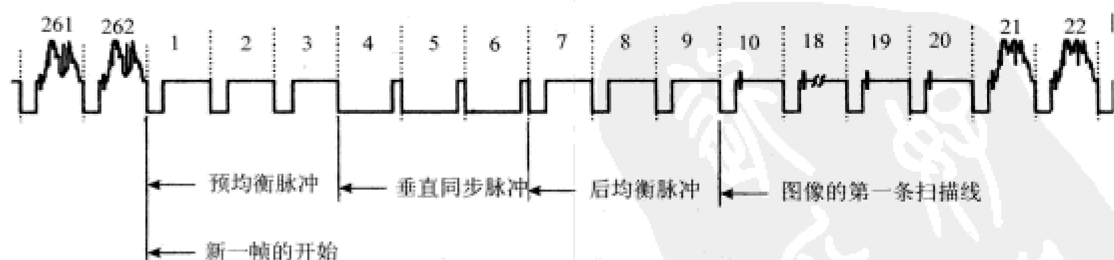


图 13-3　一个完整的视频帧信号

需要注意的是，在构成图像帧的所有扫描线中，有 3 条扫描线是由延长的同步脉冲（synchronization pulse）组成的，用于提供垂直同步信息，从而识别新一帧的开始。它们的前后又各有 3 条额外的扫描线，分别称为预均衡和后均衡扫描线。

13.4　复合视频信号的产生

如果把我们将要开发的工程范围限定为产生简单的黑白图像（没有灰度，没有颜色），并且不采用隔行扫描方法，那就可以很大程度地简化本工程所需的软硬件设备。尤其是硬件接口，可以简化成只用 3 个电阻连接到 2 个数字 I/O 引脚上即可。其中一个 I/O 引脚用于产生同步脉冲，另一个 I/O 引脚用于产生实际的亮度信号（如图 13-4 所示）。

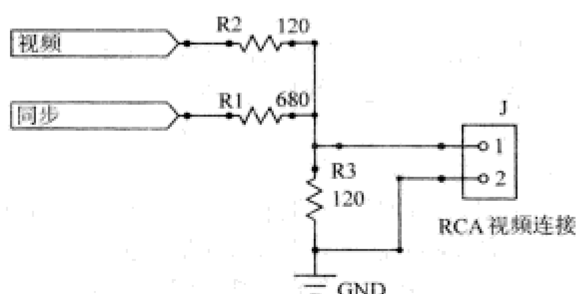


图 13-4　用于复合视频输出的简单硬件接口

所选的图 13-4 中 3 个电阻值必须使相应的亮度及同步信号的振幅接近标准配置，信号的总体振幅峰-峰值接近 1V，而电路的输出阻抗大约为 75Ω。根据前面图中所示的标准电阻值，我们可以满足这样的需求，并产生黑白图像所需的 3 种基本信号电平（如表 13-2 和图 13-5 所示）。

表 13-2　产生亮度和同步脉冲

信号特征	同　步	视　频
同步脉冲	0	0
黑色电平	1	0
白色电平	1	1

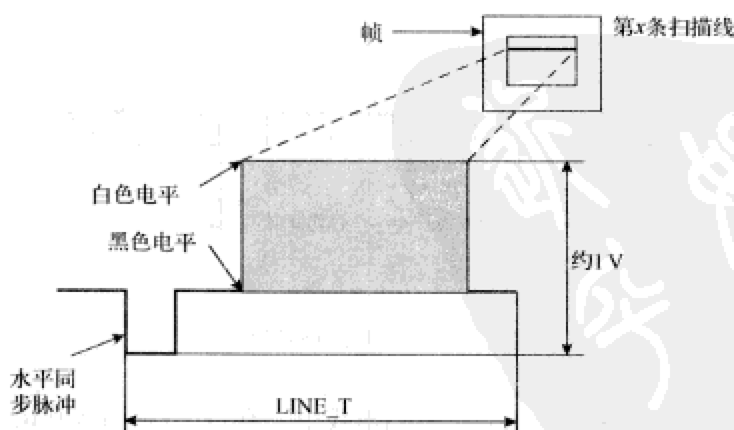


图 13-5　简化的复合视频信号

因为我们并不准备采用隔行扫描技术，所以也可以简化预均衡、垂直同步和后均衡脉冲，只需要在每个周期产生单个水平同步脉冲即可，如图 13-6 所示。

产生完整的视频输出信号现在（再一次）归结成了一个简单的状态机，它可以由定时器中断产生的固定周期来驱动。状态机非常烦琐，因为每个状态都和构成视频帧的每一种信号扫描线相关联，并且在转换到下一个状态之前会重复若干次（如图 13-7 所示）。

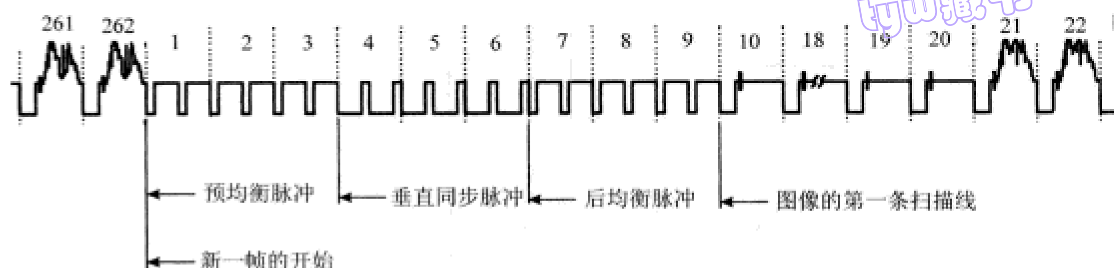


图 13-6 简化的复合视频信号（非隔行扫描）

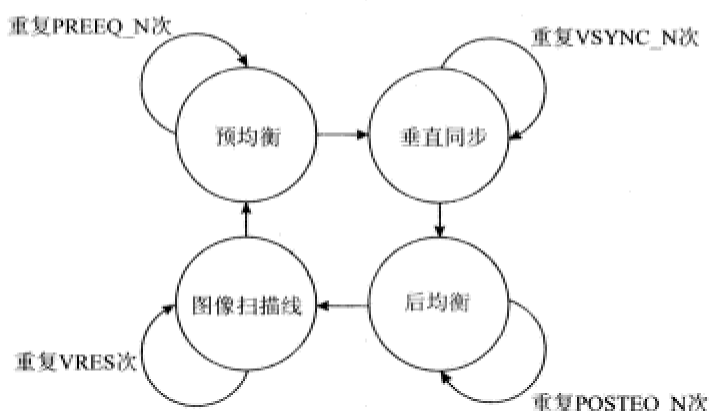


图 13-7 垂直状态机转换图

表 13-3 给出了在每个状态之间转换的描述。

表 13-3 视频状态机转换表

状 态	重 复	转 换 到
预均衡	PREEQ_N 次	垂直同步
垂直同步	3 次	后均衡
后均衡	POSTEQ_N 次	图像扫描线
图像扫描线	VRES 次	预均衡

尽管垂直同步扫描线的数量是固定的，并根据视频标准（NTSC、PAL 等）已预先设置，但是构成每帧图像的有效扫描线数量必须由用户来定义（当然应该在限定的范围内）。理论上来说可以使用所有的扫描线在屏幕上显示最大数量的视频数据信息，但是在实践中必须考虑一些限制，特别是用于存储视频图像的 PIC32 单片机的 RAM 的容量（如图 13-8 所示）。这些限制将决定用于图像扫描的扫描线的总数（VRES），而剩下的扫描线（最大为标准扫描线数量）则保留为空白。

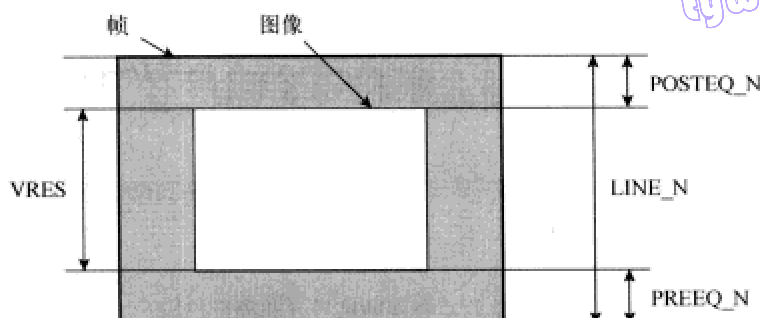


图 13-8　定义帧和图像的分辨率

如果用 LINE_N 表示组成视频帧的扫描线的总数，而用 VRES 表示希望得到的垂直分辨率，那么就可以按照如下所示来定义 PREEQ_N 和 POSTEQ_N 的值：

```
// timing for composite video vertical state machine
#ifdef NTSC
#define LINE_N .262       // number of lines in NTSC frame
#define LINE_T .2284      // Tpb clock in a line (63.5us)
#else
#define LINE_N. 312       // number of lines in PAL frame
#define LINE_T. 2304      // Tpb clock in a line (64us)
#endif

// count the number of remaining black lines top+bottom
#define VSYNC_N 3              // V sync lines
#define VBLANK_N (LINE_N -VRES -VSYNC_N)
#define PREEQ_N VBLANK_N/2            // preeq+bottom blank
#define POSTEQ_N VBLANK_N -PREEQ_N // posteq + top blank
```

如果选择 Timer 3 产生定时中断，将其时长设置为和图 13-5 中给出的水平同步脉冲周期（LINE_T）相匹配，那么就可以使用定时器关联的中断服务例程来执行垂直方向上的状态机了。以下是用于完成完整的复合视频信号刷新过程的中断服务例程的大概框架：

```
// next state table
int VS[4] = { SV_SYNC, SV_POSTEQ, SV_LINE, SV_PREEQ};
// next counter table
int VC[4] = { VSYNC_N,   POSTEQ_N,   VRES,   PREEQ_N};

void __ISR( _TIMER_3_VECTOR, ipl7) T3Interrupt( void)
{
  // advance the state machine
  if ( --VCount == 0)
  {
    VCount=VC[ VState&3];
    VState=VS[ VState&3];
  }

  // vertical state machine
  switch ( VState) {
    case SV_PREEQ:
      // horizontal sync pulse
      ...
      break;

    case SV_SYNC:
```

```
      // vertical sync pulse
      ...
      break;
    case SV_POSTEQ:
      // horizontal sync pulse
      ...
      break;
    default:
    case SV_LINE:
      ...
      break;
  } //switch
  // clear the interrupt flag
  mT3ClearIntFlag();
} // T3Interrupt
```

以下几种方式可以用于产生水平同步脉冲输出。

(1) 直接控制一个 I/O 引脚，并使用不同的延迟循环。

(2) 控制一个 I/O 引脚，使用另一个定时器（中断）来产生所需时序。

(3) 使用输出比较模块（Output Compare Module）和相应的中断服务例程。

第一种方式是最容易编码的，但是也有明显的缺陷，那就是必须让处理器一直处在永不停止的循环之中，因此在产生视频信号时也就不能执行其他任务。

第二种方式明显更加有效，并且我们现在在使用定时器和中断服务例程来执行小型状态机方面已经积累了很多的经验。

第三种方式涉及我们还没有介绍的新的外围设备模块的使用，因此需要多花费一点力气。

13.5 输出比较模块

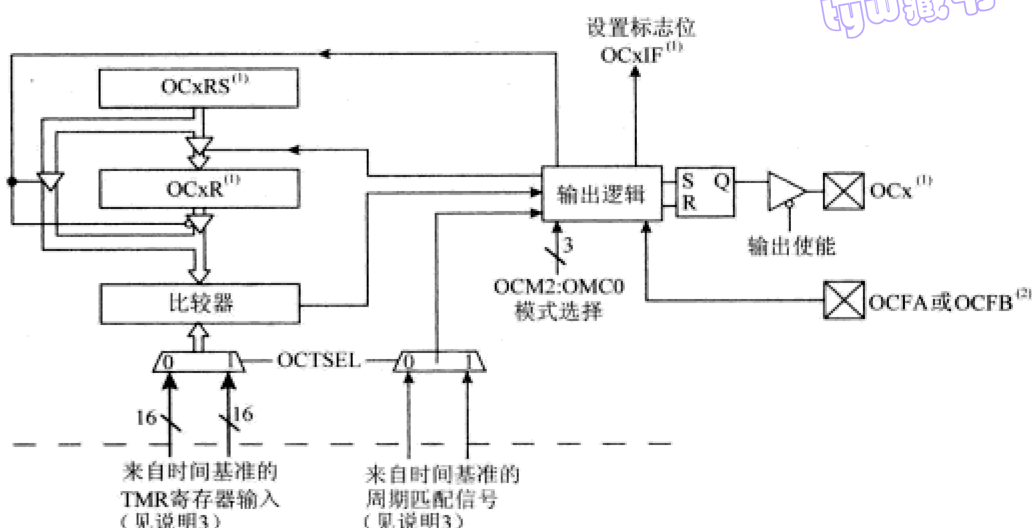
PIC32MX 系列单片机提供了 5 个输出比较模块，提供不同的应用，包括单脉冲产生、连续脉冲产生和脉冲宽度调制（PWM）。每个模块都可以关联 1 个 16 位定时器（Timer 2 或 Timer 3）或者 1 个 32 位定时器（由 Timer 2 和 Timer 3 组合而成），并且有一个输出引脚可以配置为双态模式，在需要时产生上升沿或下降沿（如图 13-9 所示）。最重要的是，每个模块都有一个相关联的独立中断向量。

输出比较模块的基本配置由 OCxCON 寄存器来进行，该寄存器有若干个控制位，其分布方式也是我们所熟悉的，用于选择所需操作方式（如图 13-10 所示）。

特别需要说明的是，当输出比较模块工作在连续缓冲模式（OCM=101）下时，OCxR 寄存器用于确定输出引脚置位的相对时间（相对于关联定时器的值），而 OCxRS 寄存器则用于确定输出引脚复位的时间（如图 13-11 所示）。

选择 OC3 模块，现在我们可以直接把输出引脚 RD2 和 Sync 输出连接在一起，如图 13-4 所示。

也可以通过垂直状态机内部的赋值来保证每个状态下 OC3 都会产生正确宽度的脉冲。尽管在常规、预均衡和后均衡扫描时，水平同步脉冲很短（大概 5μs），但是在组成垂直同步信息的 3 条扫描线上，水平同步脉冲必须变宽，从而覆盖大部分周期（参见图 13-6 中的 4、5 和 6 扫描线）。



说明 1．图中出现"x"的地方，表示与不同输出比较通道相关联的不同寄存器

2．OCFA引脚控制OC1~OC4通道

3．每个输出比较通道可以使用2个可选的时基之一。参考设备数据手册可以得知与具体模块相关联的时基是多少

图 13-9　输出比较模块框图

U-0	U-0	U-0	U-0	U-0	U-0	U-0	U-0
—	—	—	—	—	—	—	—
位 31							位 24

U-0	U-0	U-0	U-0	U-0	U-0	U-0	U-0
—	—	—	—	—	—	—	—
位 23							位 16

R/W-0	R/W-0	R/W-0	U-0	U-0	U-0	U-0	U-0
ON	FRZ	SIDL	—	—	—	—	—
位 15							位 8

U-0	U-0	R/W-0	R-0	R/W-0	R/W-0	R/W-0	R/W-0
—	—	OC32	OCFLT	OCTSEL	OCM<2:0>		
位 7							位 0

图 13-10　输出比较控制寄存器 OCxCON

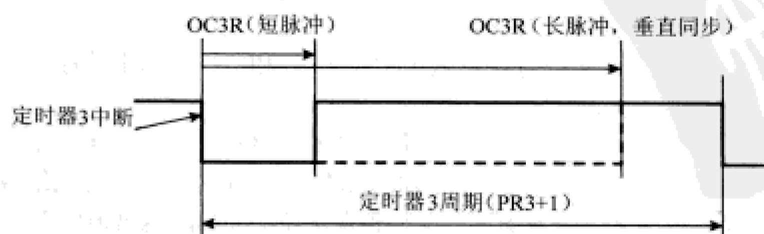


图 13-11　输出比较模块的连续脉冲模式

```
...
  // vertical state machine
  switch ( VState) {
    case SV_SYNC: // 1
      // vertical sync pulse
        OC3R=LINE_T - HSYNC_T - BPORCH_T;
        break;

    case SV_POSTEQ: // 2
      // horizontal sync pulse
        OC3R=HSYNC_T;
        break;

    case SV_PREEQ: // 0
      // prepare for the new frame
      VPtr=VA;
      break;

    default:
    case SV_LINE: // 3
      VPtr += HRES/32;
      break;
  } //switch
...
```

13.6 图像缓冲

到目前为止，我们已经学习了如何产生同步脉冲 Sync 信号，该信号通过简单的硬件接口进行连接（参考图 13-4）。出现在屏幕上的实际图像将通过加入另一种数字信号来产生。将扫描线分成不同的小段，把 Video 引脚置为双态，就能够交替变换扫描线各个小段的值，从而使其画出白色（1）和黑色（0）。因为 NTSC 标准指定最大亮度信号带宽约为 4.2MHz（PAL 也有相似的限制），前沿和后沿之间的时间距离大概为 52μs，那么黑白交替出现的小段（也就是周期数）的最大数量为 218（52×4.2），或者换句话说，最大水平分辨率的理论值是 436 像素/线（假设完全从屏幕的一边扫描到另一边）。最大垂直分辨率由组成每帧的扫描线的总数减去均衡线的最小值（6）和垂直同步线的最小值（3）得来。（对于 NTSC 标准来说，在本例中该值为 253。）

如果想产生最大图像，那它必须由一个 253×436 像素的阵列组成，也就是 110 308 个像素。如果每个像素用 1bit 来表示，一幅完整的帧图像需要分配 13.5KB 的数组，那么就用掉了 PIC32MX360 上可用 RAM 空间的 50%。在实际应用中，尽管能够产生高分辨率的输出是很好的，但是必须保证图像能够放入 RAM，同时还留有足够的空间使应用程序能顺利运行并分配给栈及变量使用。要想达到这个目的，水平分辨率和垂直分辨率值的选择方法其实有很多种，但要选择最合适的分辨率，必须考虑以下两个问题。

- 水平分辨率值是 32 的倍数，能简化确定像素在图像缓冲区中位置的计算，并能最大限度地利用单片机的 32 位总线。
- 水平分辨率和垂直分辨率的比例接近 4:3，可以避免图像出现几何畸变（屏幕上的圆形看起来更像椭圆形）。

将水平分辨率选择为 256 像素（HRES），垂直分辨率为 200 条线（VRES），可以算出所需的图像存储器是 6 400 字节（256×200/8），占用了 RAM 总量的 20%左右。使用 MPLAB C32 编译器，可以很容易分配一个整数数组（每个整数包含 32 像素）作为整个图像的存储器映射：

```
int VMap[VRES*(HRES/32)];
```

13.7　串行化、DMA 和同步

如果每条扫描线在存储器中都用 VMap 数组中的一行整数（8 个）来表示，那就必须在复合图像波形中前沿和后沿之间的短暂时间（52μs）内，按照时间顺序将每位（像素）串行化输出。换句话说，必须每 200ns 或者在更快的时间内就对选定的 Video 输出信号引脚用新的像素值进行置位或复位一次。换算来看就是两个像素之间间隔 14 个指令周期，对于简单的移位循环来说速度太快了，哪怕直接用汇编语言编程也做不到。更坏的情况是，即使假设可以把循环代码尽量压缩，最终还是得将大量的处理器时间花在视频图像产生上，那主程序能利用的处理器周期就非常少了。

幸运的是，PIC32 上有一个外围设备可以帮助我们有效地串行化图像数据，那就是 SPI 同步串行通信模块。在之前的内容中，我们使用 SPI2 模块和串行 EEPROM 设备通信。已知 SPI 模块是由一个简单的移位寄存器构成的，该寄存器可以由外部时钟信号驱动（从模式下），也可以由内部时钟信号驱动（主模式下）。现在我们将使用 SPI1 模块作为主设备，直接把 SDO（串行数据输出，RF8）引脚连接到视频硬件接口的 Video 引脚上，而 SDI（数据输入）和 SCK（时钟输出）引脚则并不使用。在 PIC32 的 SPI 模块以及 PIC32 本身的众多新奇而先进的特征当中，有两个特征特别适合本视频应用程序。

- 在 32 位模式下工作的能力。
- 可以和另一个有电力驱动的外围设备（直接存储访问（DMA）控制器）相连接。

如果工作在 32 位模式下，就可以把存储器中的图像缓冲区和 SPI 模块之间的数据传输速度提高 4 倍。通过调整 DMA 控制器的连接，还可以完全把单片机处理器从涉及视频数据的串行化工作中解脱出来。

缺点是，PIC32 的 DMA 控制器是一个功能强大而又复杂的模块，需要多达 20 个单独控制寄存器对其进行配置。但优点也有，那就是所有功能都可以通过一个同样强大而又有大量参考文档的库文件（即 dma.h）来管理，该库文件是 plib.h 的一部分。

DMA 模块和 PIC32 共用 32 位宽的系统总线，并且工作在全系统时钟频率下。它可以在 PIC32 的任意外围设备和任何存储器块之间执行任意体积的数据传输。它能够产生自己的特定中断集合，并能同时执行多个任务。也就是说，因为 4 个通道中的每一个都可以同时工作（交替访问系统总线）或串行工作（通道行为可以组成一条链表，一个通道的传输完成将发起另一个通道的传输行为）。如图 13-12 所示。

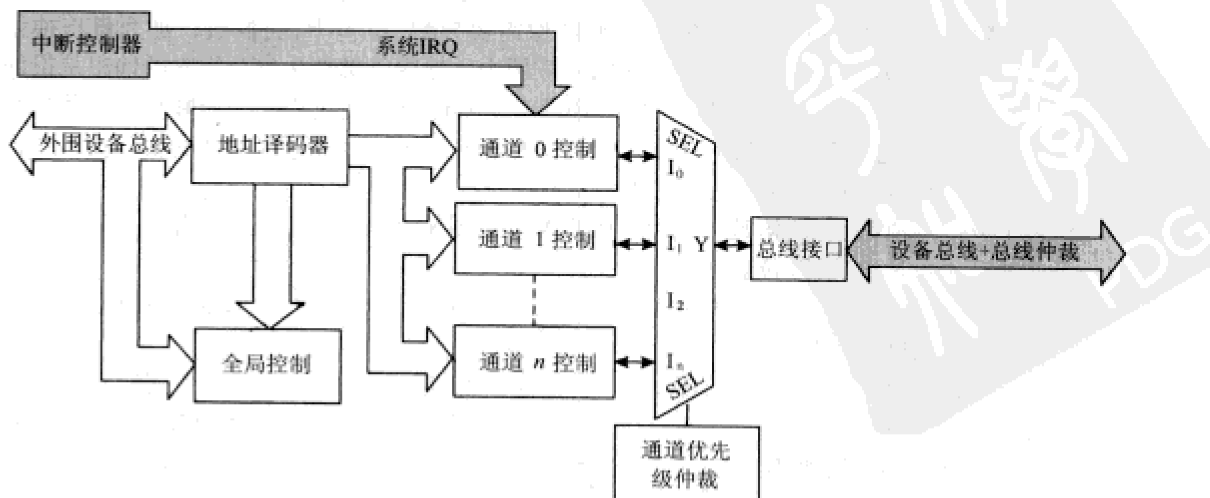


图 13-12　DMA 控制器框图

系统总线使用的仲裁由 BMX 模块来提供（之前曾经使用过这个模块），并且是无缝进行。特别要说明的是，当单片机的 Cache 系统使能、预取 Cache 有效时，仲裁行为对单片机性能的影响几乎可以忽略不计。实际上，当一个应用程序需要快速数据传输时，从性能和效率上来讲，没有比 DMA 控制器更好的选择了。

DMA 模块的初始化需要以下几个函数调用。

- DmaCHOpen()，使能 DMA 模块并准备好进行“常规”数据传输，即通常每次都需要在外围设备和存储器间传输最大 256KB 的数据，或者在存储器和存储器之间传输超过 64KB 的数据。
- DmaChnSetEventControl()，确定由哪个外围设备事件（中断）触发数据块的传输。
- DmaChnSetTxfer()，告诉控制器数据来自何方，将传向何处，以及每次传输多少字节，总共需要传输多少字节。
- DmaChnSetControl()，允许用户将多个通道的行为串起来形成一个有顺序的执行过程。

那么，假设我们初始化 DMA 控制器的通道 0 用于响应 SPI1 模块的请求（在传输缓冲区为空时产生中断），那么，针对每条 32KB 的扫描线，每次传输 32bit（4KB），必须用到以下 3 行代码：

```
DmaChnOpen( 0, 0, DMA_OPEN_NORM);
DmaChnSetEventControl( 0, DMA_EV_START_IRQ_EN |
                        DMA_EV_START_IRQ(_SPI1_TX_IRQ));
DmaChnSetTxfer( 0, (void*)VPtr, (void *)&SPI1BUF,
                HRES/8, 4, 4);
```

用户所做的所有事情就是让 PIC32 初始化第一次 SPI 传输，将第一个 32 位字写入到 SPI1 模块的数据缓冲区中（SPI1BUF），剩下的事情自动由 DMA 模块来完成。

可是这又带来了一个新的效率问题。在 Timer 3 的中断表示新的扫描时段开始和 SPI1 开始传输之间，存在大约 10μs 的时间差。对于工作在 72MHz 频率下的单片机来说，要等待这么长的时间是不可思议的（这段时间内大概可以执行 720 条有用指令），而且这个延迟时间还必须特别精确才行。哪怕只有 1 个时钟周期的差异，也会被电视机的视频同步电路放大，导致屏幕上出现看得见的“锯齿”线。更坏的情况是，如果时间差不能完全确定，而 PIC32 的 Cache 又使能（Cache 行为是非常不可预期的），还将导致屏幕左边缘出现抖动。因为我们不愿意牺牲这么关键的 PIC32 性能，所以必须找到一个办法来消除这个时间差，从而保证水平同步脉冲和 SPI 数据串行化传输的开始是绝对同步的（如图 13-13 所示）。

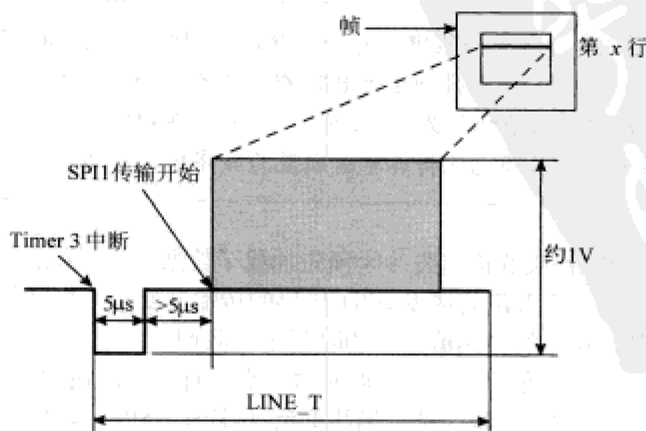


图 13-13　同步脉冲和 SPI 传输开始之间的同步

仔细观察 SPI 模块，并将它和之前的 PIC 架构中的 SPI 模块相比较，你会发现它有一个新功能，看起来可以为达到我们的目的而服务。这个功能被称为帧式从模式（Framed Slave mode），由 SPIxCON 寄存器的 FRMEN 位进行使能。不要将它和 SPI 端口的总线主操作模式和从操作模式相混淆，对 SPI 来说，实际上是增加了 2 个新的帧式操作模式。在帧式模式下，SS 引脚在不用于选择 SPI 总线上的外围设备的情况下，可以当成一个同步信号来使用。

- 选择帧式主模式时，它作为输出引脚，表示新一轮传输中的第一位。
- 选择帧式从模式时，它作为输入引脚，触发下一次数据传输的开始。

现在 SPI 端口一共可以配置成 4 种模式。

- SPI 总线主设备，帧式主设备。
- SPI 总线主设备，帧式从设备。
- SPI 总线从设备，帧式主设备。
- SPI 总线从设备，帧式从设备。

我们对第二种配置特别感兴趣，因为在这种配置下，SPI 端口是总线主设备，因此不需要在 SCK 引脚上加载外部时钟信号，而它又是帧式从设备，因此在开始数据传输之前它会等待 SS 引脚变成有效。最有用的地方在于，你将发现 SS 帧式信号的方向是可以选择的！

我们的同步问题算是完全解决了（如图 13-14 所示）。现在可以将 OC3 输出（RD2 引脚）和 SPI1 模块的 SS 输入（RB2 引脚）连接起来（直接连接或者通过一个小阻抗电阻来连接），SS 引脚为高电平有效。

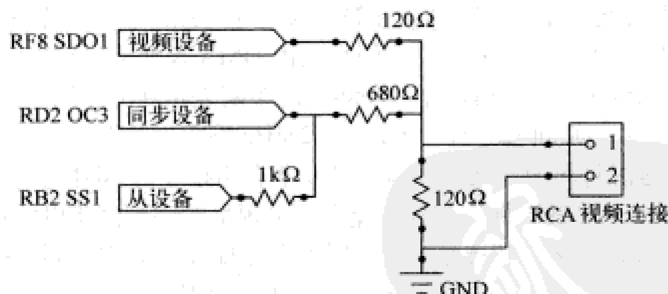


图 13-14 复合视频信号接口

> **注解** 在 RB2 引脚的众多功能中，有一个功能是用作到达 ADC 的输入通道 2 的。因此默认状态下，所有这样的引脚在上电时都被配置为模拟输入口。如果真是这样配置的，那么它的数字输入值就总为 1（高电平）。因此在把它当作有效的帧式从模式下的输入接口使用之前，必须记住将其重新配置为数字输入引脚。

假设我们已经在 SPI1 模块的输出缓冲区预先加载了数据，那么按照这种连接方式，由 OC3 模块产生的水平同步脉冲的上升沿将触发 SPI1 模块的传输开始。但是此时就开始发送扫描线数据（来自存储器的视频数据中的像素值）还为时尚早。我们必须注意后沿的时序，从而留出足够多的时间将图像放置于屏幕正中央。要想做到这一点，有一个简便的办法，就是用全零的 32 位数据预填充 SPI1 模块缓冲区中每一条扫描线的前 4 字节。SPI1 模块会依次送出数据，但是因为第一个 32 位为全零，这样就获得一点宝贵的时间，让 DMA 稍后传输实际的数据。

但是，SPI1 模块串行化一个 32 位字到底需要多少时间呢？如果单片机工作在 72MHz 时钟频率下，而外围设备时钟以 2 倍来分频，并且假设 SPI 的波特率能被 4 整除（SPI1BRG=1），那么这个串行化时间将是 3.5μs。这个值肯定低于 NTSC 标准中指定的后沿的最小值。有两种办法可以进一步增加后沿的时长。

- 在图像中再加一列，多出来的这一列总是设置为 0，并且从不用于绘制任何实际的图像。
- 使用另一个 DMA 通道，使其总是指向一个值为 0 的数组序列（越多越好），并让两个 DMA 通道自动排队运行。

这两种方法都增加了应用程序的开销，因为它们都使用了宝贵的硬件资源。加入新列意味着使用更多的 RAM，本例中具体值为 800 字节。使用另一个 DMA 通道（一共 4 个）看起来代价也很大。不过我的选择还是 DMA，因为对于我来说 RAM 永远是不够的，并且采用第二种方法，还使我们有机会体验一下 PIC32 DMA 控制器另一个很酷的功能：DMA 通道链接。

那么就会有另一个方便的函数调用，`DmaChnSetControl()`。它可以快速执行我们所需的功能，即在前一个 DMA 通道传输结束时，立刻触发下一个指定 DMA 通道的传输开始执行。以下代码说明了我们是怎么把通道 0（进行某行像素传输的那个）的执行和前一个通道 1 的执行链接起来的：

```
// chain DMA0 to completion of DMA1 transfer
DmaChnSetControl( 0, DMA_CTL_CHAIN_EN | DMA_CTL_CHAIN_DIR);
```

需要注意的是只有连续的通道可以链接起来。通道 0 只能和通道 1 链接，通道 1 则只能和通道 0 或通道 2 链接（用户决定链接方向），等等。

现在我们可以将 DMA 通道 1 配置为向 SPI1 模块传输更多值为 0 的字节，多出 4 个字节就可以将整个后沿时长变为 7μs：

```
// DMA 1 configuration  back porch extension
DmaChnOpen( 1, 1, DMA_OPEN_NORM);
DmaChnSetEventControl( 1, DMA_EV_START_IRQ_EN |
                        DMA_EV_START_IRQ(_SPI1_TX_IRQ));
DmaChnSetTxfer( 1, (void*)zero, (void *)&SPI1BUF,
                8, 4, 4);
```

这里使用的 `zero` 符号可以是一个 32 位的整数变量，需要初始化为 0；也可以是初始化为全 0 的一个整数数组，允许用户进一步延长后沿时长，并将图像置于屏幕正中。

到现在为止我们已经解决了所有关键问题，可以写一个完整的对视频生成所需的所有模块进行初始化的例程：

```
void initVideo( void)
{

   // 1. init the SPI1
   // select framed slave mode to synch SPI with OC3
   SpiChnOpen( 1, SPICON_ON | SPICON_MSTEN | SPICON_MODE32
                | SPICON_FRMEN | SPICON_FRMSYNC | SPICON_FRMPOL
                , PIX_T);

   // 2. make SS1(RB2) a digital input
   AD1PCFGSET = 0x0004;

   // 3. init OC3 in single pulse, continuous mode
   OpenOC3( OC_ON | OC_TIMER3_SRC | OC_CONTINUE_PULSE,
            0, HSYNC_T);
```

```
  // 4. Timer3 on, prescaler 1:1, internal clock, period
  OpenTimer3( T3_ON | T3_PS_1_1 | T3_SOURCE_INT, LINE_T-1);

  // 5. init the vertical sync state machine
  VState = SV_LINE;
  VCount = 1;

  // 6. init the active and hidden screens pointers
  VA = VMap1;

  // 7. DMA 1 configuration back porch extension
  DmaChnOpen( 1, 1, DMA_OPEN_NORM);
  DmaChnSetEventControl( 1, DMA_EV_START_IRQ_EN |
                         DMA_EV_START_IRQ(_SPI1_TX_IRQ));
  DmaChnSetTxfer( 1, (void*)zero, (void *)&SPI1BUF,
                  8, 4, 4);
  // 8. DMA 0 configuration image serialization
  DmaChnOpen( 0, 0, DMA_OPEN_NORM);
  DmaChnSetEventControl( 0, DMA_EV_START_IRQ_EN |
                         DMA_EV_START_IRQ(_SPI1_TX_IRQ));
  DmaChnSetTxfer( 0, (void*)VPtr, (void *)&SPI1BUF,
                  HRES/8, 4, 4);
  // chain DMA0 to completion of DMA1 transfer
  DmaChnSetControl( 0, DMA_CTL_CHAIN_EN | DMA_CTL_CHAIN_DIR);

  // 9. Enable Timer3 Interrupts
  // set the priority level 7 to use shadow register set
  mT3SetIntPriority( 7);
  mT3IntEnable( 1);
} // initVideo
```

13.8 完成一个视频库文件

现在可以完成整个视频状态机的编码，把所有必需的定义和引脚赋值都加进去：

```
/*
**    graphic.c
**    Composite Video using:
**    T3       time based
**    OC3      Horizontal Synchronization pulse
**    DMA0     image data
**    DMA1     back porch extension
**    SPI1  in Frame Slave Mode
*/
#include <p32xxxx.h>
#include <plib.h>
#include <string.h>
#include <graphic.h>

// timing for composite video vertical state machine
#ifdef NTSC
#define LINE_N  262     // number of lines in NTSC frame
#define LINE_T  2284    // Tpb clock in a line (63.5us)
#else
#define LINE_N  312            // number of lines in PAL frame
#define LINE_T  2304           // Tpb clock in a line (64us)
```

```
#endif

// count the number of remaining black lines top+bottom
#define VSYNC_N 3                        // V sync lines
#define VBLANK_N (LINE_N -VRES -VSYNC_N)
#define PREEQ_N VBLANK_N/2               // preeq + bottom blank
#define POSTEQ_N VBLANK_N -PREEQ_N       // posteq + top blank

// definition of the vertical sync state machine
#define SV_PREEQ   0
#define SV_SYNC    1
#define SV_POSTEQ  2
#define SV_LINE    3

// timing for composite video horizontal state machine
#define PIX_T     4              // Tpb clock per pixel
#define HSYNC_T   180            // Tpb clock width horizontal pulse
#define BPORCH_T  340            // Tpb clock width back porch

int VMap1[ VRES*(HRES/32)];      // image buffer
int *VA = VMap1;                 // pointer to the Active VMap

volatile int *VPtr;
volatile short VCount;
volatile short VState;

// next state table
short VS[4] = { SV_SYNC, SV_POSTEQ, SV_LINE, SV_PREEQ};
// next counter table
short int VC[4]={ VSYNC_N, POSTEQ_N, VRES, PREEQ_N};

int zero[2]= {0x0, 0x0};

void __ISR( _TIMER_3_VECTOR, ipl7) T3Interrupt( void)
{
  // advance the state machine
  if ( --VCount == 0)
  {
    VCount = VC[ VState&3];
    VState = VS[ VState&3];
  }

  // vertical state machine
  switch ( VState) {
    case SV_SYNC:     // 1
      // vertical sync pulse
      OC3R = LINE_T - HSYNC_T - BPORCH_T;
      break;

    case SV_POSTEQ:   // 2
      // horizontal sync pulse
      OC3R = HSYNC_T;
      break;

    case SV_PREEQ:    // 0
```

```
        // prepare for the new frame
        VPtr = VA;
        break;

      default:
      case SV_LINE: // 3
        // preload of the SPI waiting for SS (Synch high)
        SPI1BUF = 0;
        // update the DMA0 source address and enable it
        DCH0SSA = KVA_TO_PA((void*) VPtr);
        VPtr += HRES/32;
        DmaChnEnable( 1);
        break;
    } //switch

    // clear the interrupt flag
    mT3ClearIntFlag();

} // T3Interrupt
```

需要注意在包含实际图像数据的每一行（SV_LINE）的开始，DMA 的源指针（DCH0SSA）是如何更新从而指向下一行的像素的。在更新时，必须使用 KVA_TO_PA() 内联函数将虚拟地址转换成物理地址，该函数会进行简单的位掩码操作。你知道，DMA 控制器并不需要关心存储器空间和外围设备缓存空间之间的重新映射方式，它只需要一个物理地址。通常由 DMA 库函数来负责底层细节，我们本来也可以再一次使用 DmaChnSetTxfer() 函数来完成这项工作，但是我并没有这么做：我需要一个机会来告诉你如何直接操作 DMA 控制器的寄存器，以及如何在此过程中节省几个指令周期。

为了使其成为一个完整的图像库模块，必须再加入两个附加函数，如下所示：

```
void clearScreen( void)
{ // fill with zeros the Video array
  memset( VA, 0, VRES*( HRES/8));
} //clearScreen

void haltVideo( void)
{
  T3CONbits.TON = 0;  // turn off the vertical state machine
} //haltVideo
```

需要特别说明一下，clearScreen() 函数对于初始化图像在存储器中的映射区域（即 VMap 数组）是非常有用的。而如果有另一个重要的任务或程序需要占用 PIC32 单片机的全部处理能力时，haltVideo() 函数则能暂停产生视频信号。

把上面所示的所有函数存入文件 graphic.c 中，并把它放在 lib 目录下；我们在本章以及接下来的几章中还将使用它。同时，把这个文件加入到新的工程 Video 中。

然后，创建一个新文件，并加入以下定义：

```
/*
** graphic.h
**
** Composite video and graphic library
**
*/
#define NTSC     // comment if PAL required

#define VRES     200    // desired vertical resolution
#define HRES     256    // desired horizontal resolution pixel
```

```
void initVideo( void);

void haltVideo( void);

void clearScreen( void);
```

注意水平分辨率和垂直分辨率是在此定义的仅有的两个参数。在合理的范围内（根据时序限制以及在前面的内容中讲述的诸多考虑），可以根据特定程序的需要改变这两个参数的值，状态机以及视频产生模块中其他所有的函数也都将适应新参数的需求。

将该文件保存为 graphic.h，并将其加入到通用的 include 目录下。

13.9 测试复合视频信号

为了测试刚刚完成的复合视频模块的功能，需要使用 MPLAB SIM 仿真器，并且还需要写一个只包含几行代码的 `main()` 函数，`main()` 函数所在的文件命名为 GraphicTest.c：

```
/*
** GraphicTest.c
**
** A dark screen
**
*/
// configuration bit settings, Fcy=72MHz, Fpb=36MHz
#pragma config POSCMOD=XT, FNOSC=PRIPLL
#pragma config FPLLIDIV=DIV_2, FPLLMUL=MUL_18, FPLLODIV=DIV_1
#pragma config FPBDIV=DIV_2, FWDTEN=OFF, CP=OFF, BWP=OFF

#include <p32xxxx.h>
#include <plib.h>
#include <explore.h>
#include <graphic.h>
main()
{
  // initializations
  initEX16();     // init and enable vectored interrupts
  clearScreen(); // init the video map
  initVideo();    // start the video state machine

  // main loop
    while( 1)
    {

    } // main loop

} // main
```

记住加入 lib 目录下的 explore.c 模块，然后保存工程并使用 Build Project（生成工程）检查表来生成和链接所有模块。

打开窗口，使用逻辑分析仪检查表把 OC3 信号（同步）和 SDO1 信号（视频）加入到分析器的通道中。

此时可以运行仿真器，持续几秒钟，按下 Halt 按钮之后，转换到逻辑分析仪输出窗口观察结果（见图 13-15）。仿真器的跟踪存储区域容量是非常有限的（除非将其配置为使用扩展缓冲区），只能看到整个视频帧的小部分内容。换句话说，很有可能你会看到一个相对乏味的显示画面，仅包含几个同步脉冲序列。而 MPLAB SIM 仿真器又不能模拟 SPI 端口的输出，因此，我

们必须等待程序在真实的硬件平台上运行后才能看到正确的画面。

关于同步扫描线，我们需要观察一个有趣的时间点：那就是在每一帧的开始，采用 3 个长水平同步脉冲产生垂直同步信号时。在定时器 3 的中断服务例程里面 SV_POSTEQ 状态的第一行加一个断点，就可以在差不多新的一帧开始时让仿真暂停下来。

现在可以把窗口的中间部分放大，验证同步脉冲在预均衡、后均衡以及垂直同步扫描线中的正确性（如图 13-16 所示）。

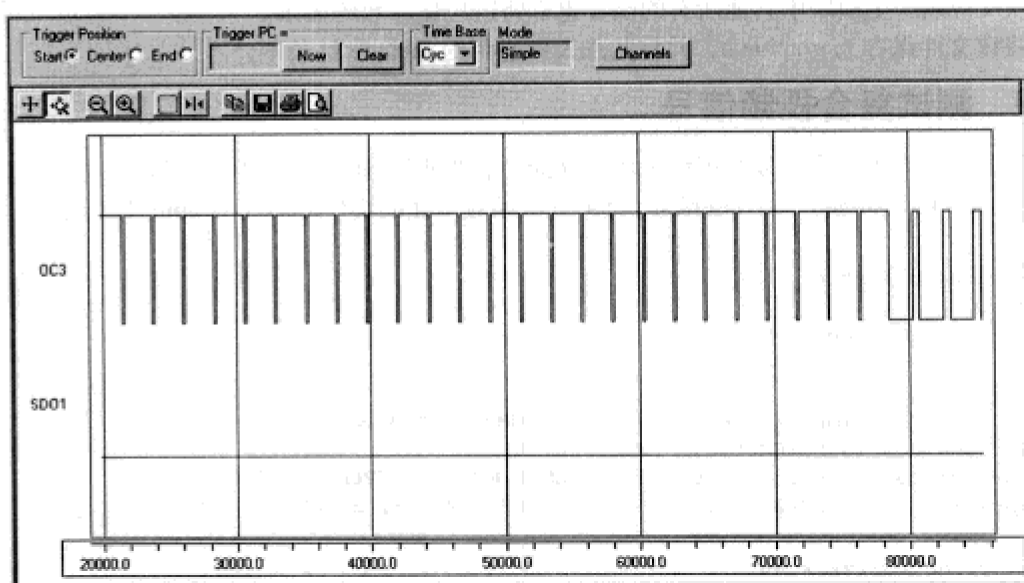


图 13-15　逻辑分析仪窗口中垂直同步脉冲的截图

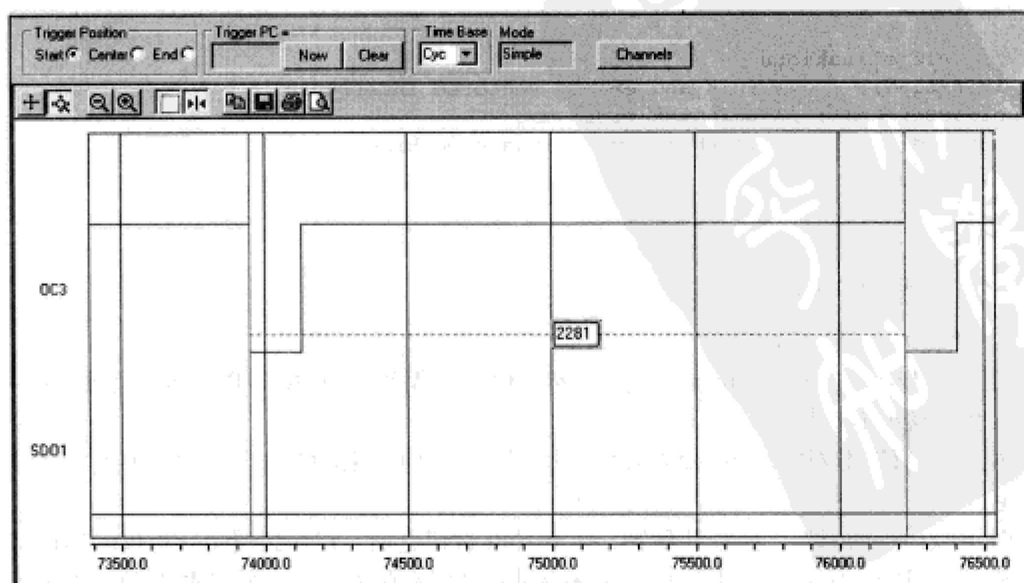


图 13-16　单个预均衡扫描线放大后的视图

记住，逻辑分析仪窗口的内容只能大概反映出相近的屏幕像素的读取，读取的准确度取决于放大的倍数（在窗口放大后会有所增加）和你的 PC 屏幕分辨率。通常来说，如果想确定一个绝对精确的时间间隔，那么最直接的办法就是将 MPLAB SIM 软件仿真器的 Stopwatch 功能和大概的断点设置结合起来使用。

13.10 测试性能

确定视频模块所需的实际处理器开销的过程会很有意思。使用逻辑分析仪我们可以直观地看到并且可以估算中断服务程序花费的处理器时间所占的比例。

和之前一样，使用 PORTA 的一个引脚（RA2）作为一个标志，当进入中断服务例程时，该位置位；执行主循环时则复位。

```
void __ISR() T3Interrupt( void)
{
_RA2=1;
...
_RA2=0;
} // T3Interrupt
```

在重新编译并且把 RA2 信号加入到逻辑分析仪工具捕获的通道中去以后（如图 13-17 所示），就可以把单条水平扫描线的扫描过程放大来观察。使用鼠标可以测试一个中断服务例程所需的大概时间。我们获取的值是 35 个周期，而整条扫描线所需的时间为 2284 个周期，这表示中断服务例程的开销不到整个处理器时间开销的 1.5%，这个显著效果要归功于 DMA 控制器的支持！

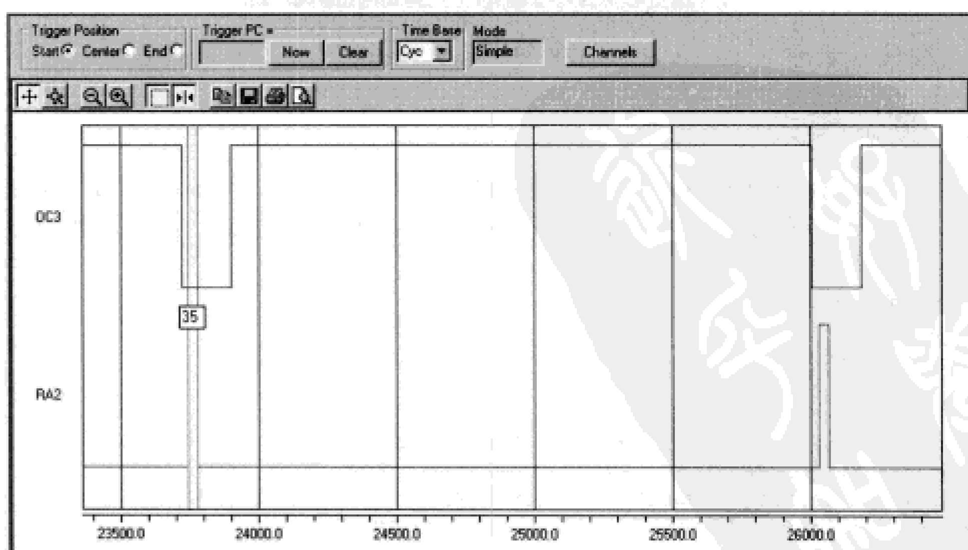


图 13-17 逻辑分析仪输出窗口的截图，测试性能

13.11 看到黑屏

用仿真器和逻辑分析仪来观察结果，在一小段时间内会很好玩，但是我相信现在你肯定渴望看到真正的视频显示！你希望在真正的电视机屏幕上，或者任何和实际的 PIC32 通过简单的

接口（只用电阻）连接起来的、可以接收复合视频信号的设备上，测试视频接口。如果你有一个 Explorer 16 演示板，那么现在就可以拿出电烙铁，在演示板右上角的原型板区上焊接 3 个电阻，并连接到标准 RCA 视频插口上。同时，如果你觉得你的电工技术很高超，那么你甚至可以开发一个小的 PCB 板作为子板（PICTail），连接到 Explorer 16 的扩展连接器上。

访问本书配套网站（www.pic32explorer.com）获取关于扩展板的更多信息，有助于你理解本书第三部分的所有高级工程。

不管你选择哪种方式，设计过程都是充满挑战的。

天，看图 13-18！事实上，当你把所有的线都接对了的时候，将 Explorer 16 演示板通电，你也只能看到一个块，按我的话来说，就是一块"黑"屏。当然，这也成功了。事实上这已经意味着大部分功能都是正确的，因为水平扫描信号和垂直扫描信号都能够被电视机进行正确的解码，从而显示出一个完美的、制式的黑色背景屏。

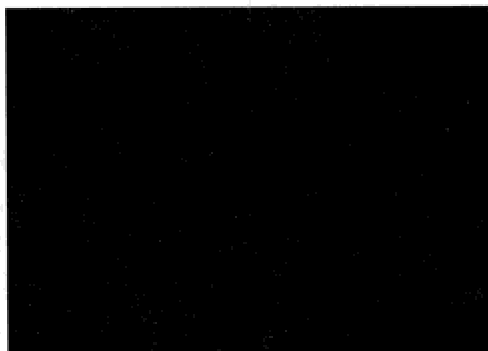


图 13-18 黑屏

13.12 测试模式

为了给我们的学习过程带来乐趣，让我们给视频数据数组填充一些值，使得视频图像有些看头，并且又很简单，可以让人立刻得知视频产生器是否工作正常。创建一个新的测试程序如下：

```
/*
** GraphicTest2.c
**
** A test pattern
**
*/
// configuration bit settings, Fcy=72MHz, Fpb=36MHz
#pragma config POSCMOD=XT, FNOSC=PRIPLL
#pragma config FPLLIDIV=DIV_2, FPLLMUL=MUL_18, FPLLODIV=DIV_1
#pragma config FPBDIV=DIV_2, FWDTEN=OFF, CP=OFF, BWP=OFF

#include <p32xxxx.h>
#include <plib.h>
#include <explore.h>
#include <graphic.h>

extern int * VA;    // pointer to the image buffer

main()
{
  int x, y;
```

```
    // initializations
    initEX16();      // init and enable vectored interrupts
    clearScreen();   // init the video map
    initVideo();     // start the video state machine

    // fill the video memory map with a pattern
    for( y=0; y<VRES; y++)
      for (x=0; x<HRES/32; x++)
        VA[y*HRES/32+x]= y;

   // main loop
      while( 1)
      {

      } // main loop
} // main
```

这次并不调用 clearScreen() 函数，而是使用两个嵌套的 for 循环来初始化 VMap 数组。外部循环（y 循环）计算扫描线数量，内部循环（x 循环）则水平移动，用扫描线的计数值来填充每一行的 8 个字（每个字 32 位）。换句话说，在第一行，每个 32 位字赋值为 0；而在第二行，每个字则被赋值为 1，以此类推直到最后一行（第 200 行），每个字被赋值为 199（十六进制表示则为 0x000000C7）。

生成新的工程来测试视频输出，可以看到如图 13-19 所示的图像模式。

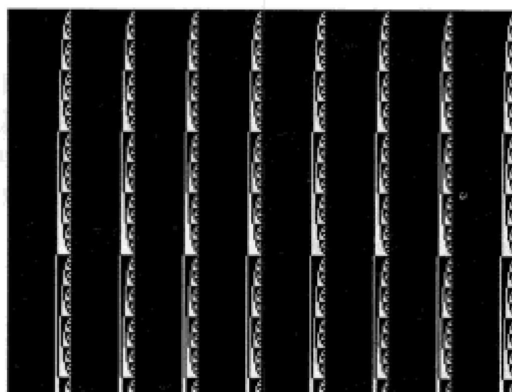


图 13-19 一种测试模式的截图

观察这个简单的测试模式，我们可以学到很多东西。首先，我们注意到每个字在屏幕上被形象化地表现为二进制的形式，最高位出现在最左边。这是 SPI 模块传输数据的顺序带来的结果：也就是说，先传输最高位。其次，可以验证最后一行包含着希望得到的模式，即 0x000000c7，于是得知存储器映射中的所有行都被显示出来了。最后，我们可以体会一下图像的细节。不同的输出设备（电视机、投影仪、LCD 显示屏等）都可以不同程度地将图像锁定，也可以根据实际的显示器分辨率和它们的输入带宽，显示一个更清晰的图像。总的来说，你可以感受到 PIC32 是如何有效地产生笔直的垂直线段的。这是一个不小的成功。

这并不意味着在最大的屏幕上就看不出输出图像上的一些小缺陷，就像微型反射波或者很小的脑电波一样的干扰信号。实际上是因为简单的 3 电阻接口就只能做到这样了。

最终来说还是整个复合视频信号接口导致了低质量的输出。你也许知道，S-Video、VGA 以及大多数其他接口都将亮度信号和同步信号分开，从而提供更稳定、更干净的画面。

13.13　绘图

现在我们已经确认图像显示模块的功能是正确的，因此可以开始将重点放在如何将它用好上。第一步自然是开发一项功能，使用户可以让屏幕上精确的坐标位置(x,y)处的像素变亮。首先需要从 y 坐标得到行数。如果 x 和 y 坐标都基于传统的笛卡尔平面坐标系表示，也就是说原点位于屏幕的左下角，那么我们需要在访问存储器映射之前，将地址进行转换，从而使存储器映射中的第一行对应于最大的 y 坐标（VRES-1）或者说 199，而最后一行则对应于 y 坐标 0。另外，因为存储器映射中每行包含 8 个字，因此需要将得到的行数乘以 32 来获取给定行的第一个字的地址。这可以用以下表达式来表示：

```
VH[ (VRES-1 -y) *8]
```

其中 VH 是指向图像缓冲区的指针。

像素按照 32 位字进行分组，因此解析 x 坐标时首先需要识别出包含该像素的字。直接用 x 除以 32 可以得到这个字在一行中的偏移。将偏移量和行地址相加，就可以得到这个字在存储器映射中的地址：

```
VH[ (VRES-1 -y)*8 + (x/32)]
```

为了优化地址计算，我们可以使用移位操作来执行如下乘法和除法：

```
VH[ ((VRES-1 -y)<<3)+(x>>5)]
```

为了识别(x, y)坐标处像素对应在 32 位字里面的那一位的具体位置，可以通过 x 除以 32 得到的余数来获取，或者采用更有效的方式，即分离出 x 坐标的最低 5 位。因为我们是想将该像素变亮，因此需要对 x 和一个合适的掩码执行二元或运算，该掩码中只有对应该像素的位置处值为 1，其他位则为 0。记住在显示时，最高位是放在最左边的（因为 SPI 模块先移出最高位），那么可以用以下表达式来计算掩码：

```
(0x80000000 >> ( x & 0x1f))
```

综上所述，我们得到绘图函数的最终表达式：

```
VH[ ((VRES-1-y)<<3)+(x>>5)] |= ( 0x80000000>>(x&0x1f));
```

最后还有一个小技巧，可以加入一个“夹子”，也就是一个简单的安全性检查，用于保证给定的坐标确实是当前屏幕范围内的有效坐标。

把下列几行代码加入到我们保存在 lib 目录下的 graphic.c 文件中：

```
void plot( unsigned x, unsigned y)
{
  if ((x<HRES) && (y<VRES) )
    VH[ ((VRES-1-y)<<3)+(x>>5)] |= ( 0x80000000>>(x&0x1f));
} // plot
```

把 x 和 y 参数定义为无符号整数，可以保证如果传递的参数是负数将会被丢弃，因为会把它们当作超出屏幕分辨率之外的大的整数。

现在将函数原型加入到 include 目录下的 graphic.h 文件中。

```
void plot( unsigned x, unsigned y);
```



注意 刚才定义的 plot() 函数是很高效的，但是不能扩展。换句话说，如果改变 graphic.h 文件中 HRES 或者 VRES 参数的值，那么就必须重新考虑如何计算给定的(x,y)坐标对应的地址以及像素对应位的位置。

13.14 一片星空

为了测试新开发的 plot() 函数，我们再一次修改 Video 工程。将 graphic.c 和 graphic.h 文件包含进来，同时使用标准 C 库中的 stdlib.h 提供的伪随机数产生器函数，产生 1 000 个随机的(x,y)坐标，我们将用下列简单代码同时测试 plot() 函数和随机数产生器的功能：

```
/*
** GraphicTest3.c
**
** A starry night
*/
// configuration bit settings, Fcy=72MHz, Fpb=36MHz
#pragma config POSCMOD=XT, FNOSC=PRIPLL
#pragma config FPLLIDIV=DIV_2, FPLLMUL=MUL_18, FPLLODIV=DIV_1
#pragma config FPBDIV=DIV_2, FWDTEN=OFF, CP=OFF, BWP=OFF
#include <p32xxxx.h>
#include <plib.h>
#include <explore.h>
#include <graphic.h>

main()
{

  int i;
  // initializations
  initEX16();      // init and enable vectored interrupts
  clearScreen();   // init the video map
  initVideo();     // start the video state machine

  for( i=0; i<1000; i++)
  {
    plot( rand()%HRES, rand()%VRES);
  }
  // main loop
    while( 1)
    {
    } // main loop
} // main
```

将文件保存为 GraphicTest3.c 并将其加入到 Video 工程中，替换之前的程序。再次生成工程并用在线调试器对 Explorer 16 演示板进行编程，视频显示输出看起来将是一片美丽的星空，就像图 13-20 中的屏幕截图显示的那样。

确实是一片星空，但并不是真实的。因为没有一个识别得出来的带状区域，能看到在它附近的星星密度明显增加——换句话说，就是没有银河！

这是一件好事！这意味着伪随机数产生器正是按照预先设定的那样在工作。



图 13-20　屏幕截图：绘出美丽星空

13.15　画出一条线

下一步就是画线，或者应该说是画出线段。显然，水平线段和垂直线段都是没有问题的，一个简单的 for 循环就可以搞定。但是画出一条斜线则完全是另外一回事。我们可以从读书时候就学过的两点之间的线段的基本公式开始：

```
Y=y0+(y1-y0)/(x1-x0)*(x-x0)
```

其中(x0, y0)和(x1, y1)分别是直线上的任意两点的坐标。

该公式给出了对于任何给定的 x 值对应的 y 坐标。因此我们可以将其用在一个循环中，用来计算线段的开始位置与结束位置之间的每个离散的 x 值所对应的 y 值，如下所示：

```
/*
** LineTest1.c
**
** testing the basic line drawing function
*/
// configuration bit settings, Fcy=72MHz, Fpb=36MHz
#pragma config POSCMOD=XT, FNOSC=PRIPLL
#pragma config FPLLIDIV=DIV_2, FPLLMUL=MUL_18, FPLLODIV=DIV_1
#pragma config FPBDIV=DIV_2, FWDTEN=OFF, CP=OFF, BWP=OFF
#include <p32xxxx.h>
#include <plib.h>
#include <explore.h>
#include <graphic.h>

main()
{
   int x;
   float x0 = 10.0,   y0 = 20.0;
   float x1 = 200.0,  y1 = 150.0;
   float x2 = 20.0,   y2 = 150.0;

// initializations
initEX16();      // init and enable vectored interrupts
clearScreen();   // clear the image buffer
initVideo();     // start the video state machine
```

```
// draw an oblique line (x0,y0) - (x1,y1)
for( x = x0; x<x1; x++)
  plot( x, y0 + (y1-y0)/(x1-x0)* (x-x0));

// draw a second (steeper) line (x0,y0) - ( x2,y2)
for( x = x0; x<x2; x++)
  plot( x, y0+(y2-y0)/(x2-x0)* (x-x0));

// main loop
  while( 1)
  {
  } // main loop

// main // main
```

产生的输出（图 13-21）仅对第一条线（较低的那条）来说，是一条可接受的连续线段，其水平距离（x1-x0）大于垂直距离（y1-y0）。在第二条线（即更高的那条）上，点与点之间看起来是不连续的，我们显然对这个结果不满意。另外，我们不得不执行浮点算术运算，比起整数运算来说，其计算量增加数倍，在前一章我们已经知道了这一点。

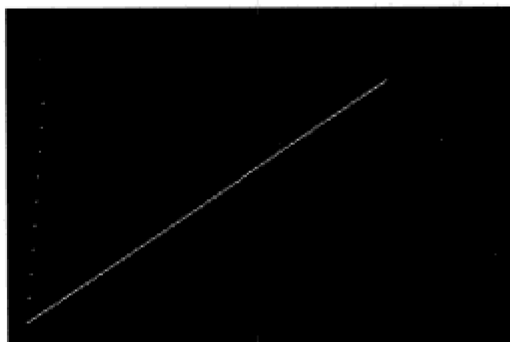


图 13-21 屏幕截图：画出斜线

13.16 Bresenham 算法

回到 1962 年，当时工作在 IBM 公司圣何塞开发实验室的 Jack E. Bresenham 发明了一个大量使用整数算术运算的画线算法。直到今天这个算法仍然被认为是所有计算机图形程序的基础。该方法基于 3 个优化“小技巧”。

(1) 将绘画方向简化成一个个别情况（从左到右）。

(2) 将直线的斜率简化成一个个别情况，即水平距离最大。

(3) 将等式两端的表达式都乘以水平距离（deltax）来获得整数值。

这样得到的画线代码非常紧凑而高效，以下是适合我们视频模块的算法：

```
#define abs( a) (((a)> 0) ? (a) : -(a))

void line( short x0, short y0, short x1, short y1)
{

    short steep, t ;
    short deltax, deltay, error;
```

```
    short x, y;
    short ystep;

  // simple clipping
    if (( x0 < 0) || (x0 > HRES))
      return;
    if (( x1 < 0) || (x1 > HRES))
      return;
    if (( y0 < 0) || (y0 > VRES))
      return;
    if (( y1 < 0) || (y1 > HRES))
      return;
    steep = ( abs(y1 - y0) > abs(x1 - x0));

    if ( steep )
    { // swap x and y
      t = x0; x0 = y0; y0 = t;
      t = x1; x1 = y1; y1 = t;
    }
    if (x0 > x1)
    { // swap ends
      t = x0; x0 = x1; x1 = t;
      t = y0; y0 = y1; y1 = t;
    }

    deltax = x1 - x0;
    deltay = abs(y1 - y0);
    error = 0;
    y = y0;

    if (y0 < y1) ystep = 1; else ystep = -1;
    for (x = x0; x < x1; x++)
    {
      if ( steep) plot(y,x); else plot(x,y);
      error += deltay;
      if ( (error<<1) >= deltax)
      {
        y += ystep;
        error -= deltax;

      } // if
    } // for

} // line
```

可以将该函数加入到视频模块 graphic.c 中，并在头文件 graphic.h 中声明其原型：

```
void line( short x0, short y0, short x1, short y1);
```

为了测试 Bresenham 算法的效率，我们创建一个新的小工程，并再一次使用伪随机数生成器函数。以下的示例代码将在屏幕上画出一个框，然后在随机产生的坐标上画出 100 条线，测试画线函数的正确性。

```
/*
** Bresenham.c
**
** Fast line drawing algorithm example
*/
// configuration bit settings, Fcy=72MHz, Fpb=36MHz
```

```
#pragma config POSCMOD=XT, FNOSC=PRIPLL
#pragma config FPLLIDIV=DIV_2, FPLLMUL=MUL_18, FPLLODIV=DIV_1
#pragma config FPBDIV=DIV_2, FWDTEN=OFF, CP=OFF, BWP=OFF

#include <p32xxxx.h>
#include <plib.h>
#include <explore.h>
#include <graphic.h>

main()
{

  int i;

  // initializations
  initEX16();      // init and enable vectore interrupts
  initVideo();     // start the state machines

  // main loop
    while( 1)
    {
    clearScreen();
    line( 0, 0, 0, VRES-1);
    line( 0, VRES-1, HRES-1, VRES-1);
    line( HRES-1, VRES-1, HRES-1, 0);
    line( 0, 0, HRES-1, 0);

    for( i=0; i<100; i++)
      line( rand()%HRES, rand()%VRES,
         rand()%HRES, rand()%VRES);
   // wait for a button to be pressed
   getKEY();
   } // main loop

} // main
```

主循环也使用了 getKey() 函数，这是我们在前一章里面开发的函数，它已被加入到 explore.h 模块中了，这样做是为了在按钮被按下之后才清除屏幕，然后在屏幕上画出新的 100 条随机线段（如图 13-22 所示）。

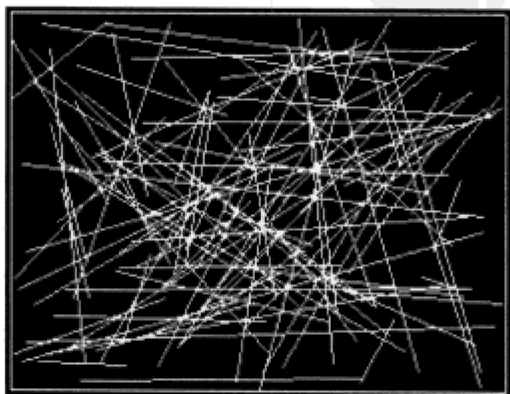


图 13-22　屏幕截图：Bresenham 画线算法测试

你会对画线算法的速度留下很深的印象。哪怕增加线段数量到1000条，PIC32的性能优势仍然是很明显的。

13.17 画出数学函数

随着图像模块的逐渐完善，我们现在可以充分利用其可视化功能的优点，来研究一些有趣的应用。一个经典的应用就是在传感器记录的数据基础上画出一幅图，或者为了演示目的而采用更简单的办法，即从给定的数学函数快速算出数据来画图。

例如，我们假设函数是一个正弦曲线函数（带弯曲），如下所示：

```
y(x) = x * sin( x)
```

同时假设我们想画的图中，x的值从 0 到 8*PI 之间变化。

稍微改动一下，就可以把该函数缩放到适合我们的屏幕，重新将输入值范围映射到 0~200，而输出值范围则为+75~-75。

以下示例程序将先在屏幕上画出x轴和y轴，然后再画出该函数的曲线。

```
/*
** graph1d.c
**
** Plotting a function graph
*/
// configuration bit settings, Fcy=72MHz, Fpb=36MHz
#pragma config POSCMOD=XT, FNOSC=PRIPLL
#pragma config FPLLIDIV=DIV_2, FPLLMUL=MUL_18, FPLLODIV=DIV_1
#pragma config FPBDIV=DIV_2, FWDTEN=OFF, CP=OFF, BWP=OFF
#include <p32xxxx.h>
#include <plib.h>
#include <explore.h>
#include <graphic.h>
#include <math.h>

#define X0 10
#define Y0 (VRES/2)

main( void)
{
  int x, y;
  float xf, yf;

  // initializations
  initEX16();      // init and enable vectored interrupts
  clearScreen();
  initVideo();     // init video state machine
 // draw the x and y axes crossin in (X0,Y0)
 line( X0, 10, X0, VRES-10);    // y axes
 line( X0-5, Y0, HRES-10, Y0);  // x axes

 // plot the graph of the function for
 for( x=0; x<200; x++)
 {
   xf = (2 * M_PI / 50) * (float) x;
   yf = 75.0 / ( 8 * M_PI) * xf * sin( xf);
   plot( x+X0, yf+Y0);
```

```
    }

    // main loop
    while( 1);

} // main
```

注意必须包含 math.h 库文件，才能使用 `sin()` 函数原型以及其他一些有用的定义，其中有π（`M_PI`）的值。

将该文件保存为 graph1d.c，替换 Video 工程中的主模块。生成工程，并用在线调试器对 Explorer 16 演示板进行编程。快一点儿，新的函数图像就要出现在屏幕上了（如图 13-23 所示）！

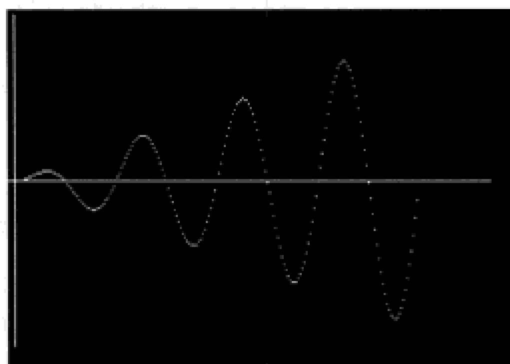


图 13-23 屏幕截图：正弦曲线函数图

如果图上的点看起来太稀疏了，那还可以使用画线算法将每个点和他之前的点连接起来。

13.18 画出二维函数图

画出二维函数曲线图更有意思。处理透视变形让你更兴奋，同时也带来了挑战，因为要把函数计算值连接起来形成形象的网格图案。

最简单的方法是在二维图像中画出三维坐标轴，形成通常所谓的正等侧投影，该方法需要的计算资源最少，并且可见变形也很小。下面的公式用于计算三维空间中一个点的坐标(x,y,z)对应在二维空间（我们的视频屏幕）中的投影坐标（px 和 py）（如图 13-24 所示）。

```
px = x + y/2;
py = z + y/2;
```

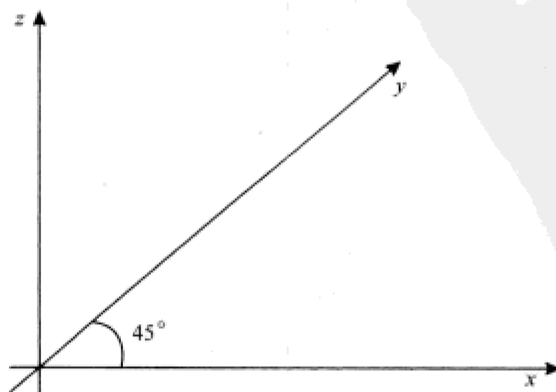


图 13-24 正等侧投影

为了画出给定函数 `z = f(x,y)` 的三维视图，我们使用两个嵌套 `for` 循环在 *x* 和 *y* 平面中画出距离相等的点组成的网格。对于每一个点，通过函数计算其 *z* 坐标，然后采用正等侧投影获得 `(px,py)` 坐标。再将新计算的点和同一行前一列上的前一个点用线段连接起来，并把这个点和同一列前一行上的前一个点用线段连接起来（如图 13-25 所示）。

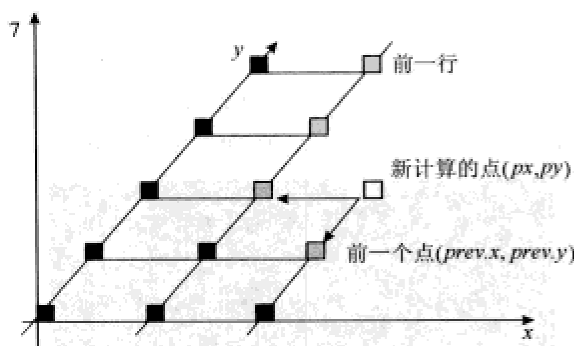


图 13-25　画出一个网格来提高三维视图的可视化效果

记录同一行上前一个计算点的坐标以及前一行上点的坐标，并不是很难的事情，但是却需要很大的存储空间。假设我们使用 20×20 的网格，那么就需要存储 400 个点的坐标。每个点用 2 个整数来存储，那就多加了 800 个字（3200 字节）的宝贵空间。实际上，从之前的绘图示例中可以确知，所有真正需要的只是当前所画网格的“边缘”上的点的坐标。因此，只需加一些判断，就可以把存储器需求降低到用一个小型（滚动）缓冲区来存储 20 个坐标对即可。

以下代码实现了这个函数的曲线图绘制：

```
z(x,y) = 1/ sqrt( x2 + y2) * cos ( sqrt( x2 + y2)
```

其中，*x* 和 *y* 的范围从 `-3*PI` 到 `+3PI`：

```
/*
** graph2d.c
**
** 07/02/06 v1.0 LDJ
** 11/21/07 v2.0 LDJ PIC32 porting
*/

// configuration bit settings
#pragma config POSCMOD=XT, FNOSC=PRIPLL
#pragma config FPLLIDIV=DIV_2, FPLLMUL=MUL_18, FPLLODIV=DIV_1
#pragma config FWDTEN=OFF, CP=OFF, BWP=OFF

#include <p32xxxx.h>
#include <explore.h>
#include <graphic.h>
#include <math.h>

#define X0       10      // graph offset
#define Y0       10
#define NODES    20      // define grid
#define SIDE     10
#define STEP     1       // movement increment

typedef struct {
    int x;
```

```
    int y;
  } point;

point edge[NODES], prev;

main( void)
{
  int i, j, x, y, z;
  float xf, yf, zf, sf;
  int px, py;
  int xoff, scale;

  // initializations
  initEX16();
  clearScreen();
  initVideo();

  xoff = 100;

  scale = 75;

  while (1)
  {
    // clear hidden screen
    clearScreen();

// draw the x, y and z axes crossing in (X0,Y0)
line( X0, 10, X0, 10);                // z axis
line( X0-5, Y0, HRES-10, Y0);         // x axis
line( X0-2, Y0-2, X0+120, Y0+120);    // y axis

// init the array of previous egde points
for( j=0; j<NODES; j++)
{
  edge[j].x = X0+ j*SIDE/2;
  edge[j].y = Y0+ j*SIDE/2;
}

// plot the graph of the function for
for( i=0; i<NODES; i++)
{
  // transform the x range to 0..200 offset 100
  x = i * SIDE;
  xf = (6 * M_PI/200) * (float)(x-xoff);
  prev.y = Y0;
  prev.x = X0 + x;

  for ( j=0; j<NODES; j++)
  {
    // transform the y range to 0..200 offset 100
    y = j * SIDE;
    yf = (6 * M_PI / 200) * (float)(y-100);

    // compute the function
    sf = sqrt( xf * xf + yf * yf);
    zf = 1/(1+ sf) * cos( sf );

    // scale the output
    z = zf * scale;
```

```
// apply isometric perspective and offset
px = X0 + x+ y/2;
py = Y0 + z + y/2;

// plot the point
plot( px, py);
    // draw connecting lines to visualize the grid
    line( px, py, prev.x, prev.y);
    line( px, py, edge[j].x, edge[j].y);

    // update the previous points
    prev.x = px;
    prev.y = py;
    edge[j].x = px;
    edge[j].y = py;
  } // for j
} // for i

// wait for a button
getKEY();

  } // main loop
} // main
```

将文件保存为 graph2d.c，替换 Video 工程中的主文件。重新生成工程并对 Explorer 16 演示板进行编程，可以看到，尽管该函数需要对 400 个点依次进行计算，并且要进行大量的浮点运算，从而在屏幕上画出多达 800 条线段，但 PIC32 还是可以很快地生成输出图像（如图 13-26 所示）。

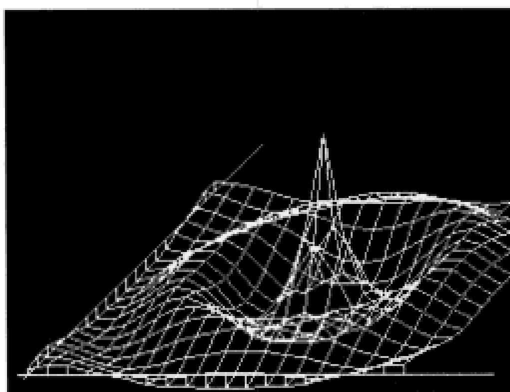


图 13-26　屏幕截图：二维函数曲线图的绘制

13.19　分形

分形（fractal）是 Benoit Mandelbrot 创造的术语，他是一个数学家，也是 IBM 公司太平洋西北实验室的在职研究人员。1975 年他提出一个数学物体的集合，这个集合呈现出一个有趣的特性：不论缩放的倍数如何，这种物体的图案总是自相似（self-similar）的，就像是用无穷次递归过程构造出来的一样。在自然界有很多分形形态存在，只是它们的自相似特性通常是以有限的标度来延伸的。这样的例子包括云朵、雪花、山川、河流脉络，还有人体中的血管。

最著名的数学分形对象示例恐怕就是 Mandelbrot 集合了，因为它赋予自身以令人印象深刻的电脑视觉效果。Mandelbrot 集合定义为复平面中的一个子集，由二次函数 z^2+c 迭代而成。通过排除，所有使上述迭代序列产生的结果不趋向于无穷大的复平面上的点 c，都被认为属于 Mandelbrot 集合。很容易验证：一旦 z 的模大于 2，迭代结果必然会发散（趋向于无穷大）。这样一来给定的点就不属于 Mandelbrot 集合了，那就可以将它排除在外，继进行下一个点的迭代。问题在于，只要 z 的模值一直小于 2，那就没有办法确定什么时候停止迭代并说明该点是属于 Mandelbrot 集合的。因此，实现 Mandelbrot 集合的计算机算法通常都设置一个任意值的最大迭代次数，并假定超过这个迭代次数以后，迭代产生的点就一定属于 Mandelbrot 集合。

以下代码说明了内部循环是怎么用 C 语言来实现的：

```
// initialization
  x = x0;
  y = y0;
  k = 0;

  // core iteration
  do {
    x2 = x*x;
    y2 = y*y;
    y = 2*x*y+y0;
    x = x2-y2+x0;
    k++;
    } while ( (x2 + y2 < 4) && ( k < MAXIT));

// check if the point belongs to the Mandelbrot set
if ( k == MAXIT) plot( j, i);
```

其中，x0 和 y0 是复平面上点 c 的坐标。

我们可以对复平面上的某个方形子集中的每一个点重复该迭代，从而获取整个 Mandelbrot 集合的图形。对 c 的模值的考虑意味着整个集合必须包含在以原点为中心而半径为 2 的圆盘内，因此，就像我们在开发第一个程序时那样，我们要在包含 HRES×VRES 个点的网格内对复平面进行扫描（充分利用视频模块的全屏分辨率），从而保证整个圆盘能够在屏幕上显示：

```
/*
** Mandelbrot.c
**
** Mandelbrot Set graphic demo
*/
// configuration bit settings, Fcy=72MHz, Fpb=36MHz
#pragma config POSCMOD=XT, FNOSC=PRIPLL
#pragma config FPLLIDIV=DIV_2, FPLLMUL=MUL_18, FPLLODIV=DIV_1
#pragma config FPBDIV=DIV_2, FWDTEN=OFF, CP=OFF, BWP=OFF

#include <p32xxxx.h>
#include <plib.h>
#include <explore.h>
#include <graphic.h>

#define SIZE VRES
#define MAXIT 64
```

```
void mandelbrot( float xx0, float yy0, float w)
{
  float x, y, d, x0, y0, x2, y2;
  int i, j, k;
  // calculate increments
  d = w/SIZE;
  // repeat on each screen pixel
  y0 = yy0;
  for (i=0; i<SIZE; i++)
  {
    x0 = xx0;
    for (j=0; j<SIZE; j++)
    {
      // initialization
        x = x0;
        y = y0;
        k = 0;

      // core iteration
        do {
          x2 = x*x;
          y2 = y*y;
          y = 2*x*y + y0;
          x = x2-y2 + x0;
          k++;
        } while ( (x2 + y2 < 4) && ( k < MAXIT));

      // check if the point belongs to the Mandelbrot set
      if ( k == MAXIT) plot( j, i);

      // compute next point x0
      x0 += d;
    } // for j
    // compute next y0
    y0 += d;
  } // for i
} // mandelbrot

int main( void)
{
  float x, y, w;
  int c;
  // initializations
  initEX16();       // init and enable vectored interrupts
  initVideo();      // init the video state machine
  // intial coordinates lower left corner of the grid
  x = -2.0;
  y = -2.0;
  // initial grid size
  w = 4.0;

  clearScreen();          // clear the screen
  mandelbrot( x, y, w);// draw new image

  while( 1);
} // main
```

将该文件保存为 Mandelbrot.c 并把它加入到一个新的名为 Mandelbrot 的工程中。确定其他所有需要的模块都已经加入到工程中，包括 graphic.c、graphic.h 和 explore.c。生成该工程，使用在线调试器对 Explorer 16 演示板进行编程。如果一切进展顺利，那么运行程序之后，就可以看到所谓的 Mandelbrot “心脏线”显示在屏幕上（如图 13-27 所示）。

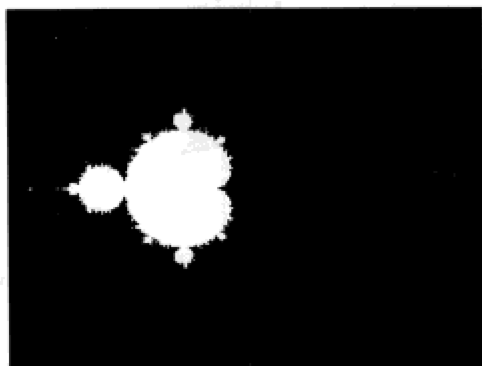


图 13-27　屏幕截图：Mandelbrot 心脏线

在我还是个孩子的时候，我买了第一台个人电脑（实际上，家用计算机的说法那时候已经开始了，用于称呼 Sinclair ZX Specturm），那时候我就开始玩分形程序了。所以我清晰地记得我曾经几个小时都盯着电脑屏幕，等那台老旧却又可靠的 ZX80 处理器（工作在强大的 3.5MHz 主频下）画出这样的图案。几年以后，我又买了一台 IBM PC，这是一台采用 8088 处理器、运行在主频高不了多少的 4MHz 下的 XT 的翻版，因而性能也好不了多少。并且，尽管我的单色 Hercules 显卡的分辨率更高一些，但是我仍然必须在夜里就启动程序，而直到第二天早上才能看到结果，处理时间有时候长达 8 个小时。

> 很显然，绘制分形图案所需的计算能力随着所选区域和最大迭代次数（MAXIT）的不同而有显著不同。但是，尽管我也看过该程序在其他处理器上的运行情况，包括在 PIC24（32MHz）上的运行情况，但是在我第一次看到 PIC32 在不到 5 秒钟的时间内就画出了心脏线时，我仍然激动万分！

真正的快乐才刚刚开始。Mandelbrot 集合最有趣的地方在于它图案的边缘部分，我们可以将其边缘放大和缩小，从而可以发现一个无限复杂的世界。我们不仅仅只观察属于 Mandelbrot 集合的那些点，同时也观察那些在边缘就发散了的点，并给每个点按照发散的速度进行着色，就可以进一步提高图案的艺术性。因为我们仅仅使用单色显示，因此只能简单地根据每个点到达最大模值或者到达最大迭代次数之前所进行的迭代次数，给每个点用黑白两色交替着色。采用这么简单的着色方法，也意味着只需要对前面的代码改动一行即可：

```
// check if the point belongs to the Mandelbrot set
if ( k & 2) plot( j, i);
```

另外，因为把玩 Mandelbrot 集合的最好方式就是选择新的区域，然后放大细节，所以我们可以修改主程序循环，从而通过按下 Explorer 16 演示板上的某个按钮来选择图像的某一部分。我们可以想象一下把整个图像分成 4 个相应的正方形，从左上方开始按照顺时针方向编号，并通过等分网格区域（w）来获取双倍的分辨率（如图 13-28 所示）。

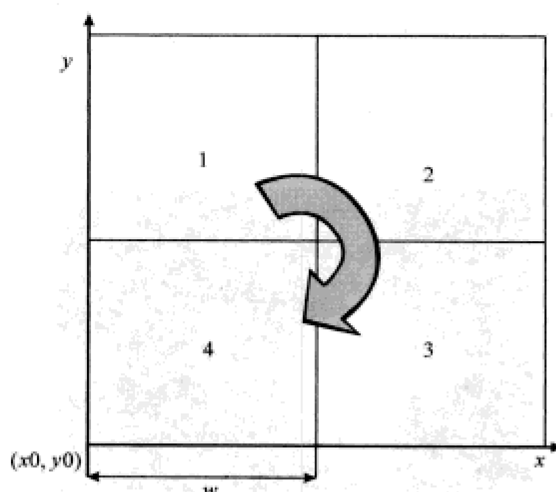


图 13-28　将屏幕分成 4 个正方形

```
int main( void)
{
  float x, y, w;
  int c;

  // initializations
  initEX16();
  initVideo();      // start the state machines

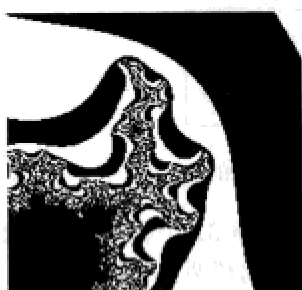
  // intial coordinates lower left corner of the grid
  x = -2.0;
  y = -2.0;
  // initial grid size
  w = 4.0;

  while( 1)
  {
    clearScreen();          // clear the screen
    mandelbrot( x, y, w);// draw new image
    // wait for a button to be pressed
    c = getKEY();
    switch ( c){
    case 8:   // first quadrant
      w/= 2;
      y += w;
      break;
    case 4:   // second quadrant
      w/= 2;
      y += w;
      x += w;
      break;

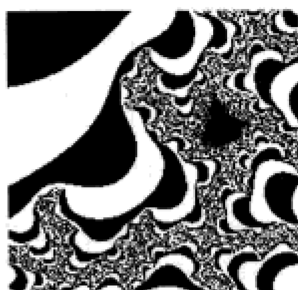
    case 2:   // third quadrant
      w/= 2;
      x += w;
      break;
```

```
    default:
    case 1:    // fourth quadrant
      w/= 2;
      break;
    } // switch
  } // main loop
} // main
```

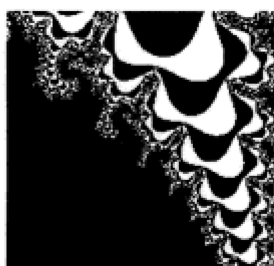
图 13-29 显示了选择的一块有趣的区域，你可能需要一点耐心来好好研究它。



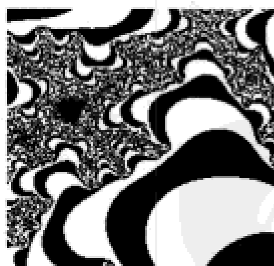
(a) (+0.25+j 0.5) w=0.25



(b) (+0.37500−j 0.57813) w=0.01563



(c) (−1.28125+j 0.3125) w=0.3125



(d) (+0.34375+j 0.56250) w=0.3125



(e) (−1.28125+ j 0.4688)w=0.01563

图 13-29 有趣的图案

13.20 文本

到目前为止我们主要工作都集中于简单图案的可视化，但是你肯定不止一次地希望能够在屏幕上显示一些文本内容。在视频存储器中写入文本信息和画点或者画线其实没什么两样，实际上，可以采用很多方法来做到这一点，包括通过我们刚刚开发的画点和画线函数来进行。但是为了获取更好的性能和体积更小的代码，在图像显示屏上绘制文本的最简单的办法还是开发一种固定间距的字体。每个字符可以画在 8×8 的像素格子里面，这样的话，一字节就可以编码一行，而 8 字节就可以编码整个字符了。然后就可以组成一个由字母、数字和标点符号组成的基本集合，按照它们出现在 ASCII 字符集中的顺序进行排列，用单个 char 类型的数组就可以形成一个简单的字体（如图 13-30 所示）。

0	0	0	1	1	1	0	0
0	0	1	0	0	0	1	0
0	0	1	0	0	0	1	0
0	0	1	1	1	1	1	0
0	0	1	0	0	0	1	0
0	0	1	0	0	0	1	0
0	0	1	0	0	0	1	0
0	0	0	0	0	0	0	0

图 13-30 用简单的 8×8 字体表示的字母 A

为了节省空间，我们不需要创建 ASCII 字符集中的前 32 个代码，因为它们大部分对应着一些命令和特殊的同步码，都是过去的电传打字机和调制解调器才使用的。

```
/*
** 8 x 8 Simple Character Font
**
*/
#define F_OFFS   0x20 // initial offset
#define F_SIZE   96 // define only the first 96 characters

const char Font8x8[]={
// 20 - SPACE
    0x00,        // 0b 0000000,
    0x00,        // 0b 0000000,
    0x00,        // 0b 0000000,
    0x00,        // 0b 0000000,
    0x00,        // 0b 0000000,
    0x00,        // 0b 0000000,
    0x00,        // 0b 0000000,
    0x00,        // 0b 0000000,
// 1 - !
    0x18,        // 0b 0011000,
    0x18,        // 0b 0011000,
    0x18,        // 0b 0011000,
    0x18,        // 0b 0011000,
    0x18,        // 0b 0011000,
    0x00,        // 0b 0000000,
    0x18,        // 0b 0011000,
    0x00,        // 0b 0000000,
...
} // Font 8x8[]
```

注意 Font8x8[]数组定义为 const，因为其内容在程序执行过程中几乎保持不变，并且最好是给它分配 PIC32 的 Flash 存储器空间，从而节省宝贵的 RAM 空间。

当然，每个字符形状的定义是可以依据个人喜好而定的。欢迎你更改 Font8x8[]数组的内容来迎合自己的口味。



注解 定义一种新的字体是一项冗长而细致的工作，但是也是一个有很大创作空间的工作。我知道有些读者会享受其中的乐趣。把 font.h 文件中的完整字体列出来会浪费本书好几页的空间，因此我决定省略不写。你可以在本书附带资源中找到该文件。

在屏幕上打印一个字符现在就变成了把 8 字节从字体数组中复制到屏幕上相应的位置。最简单的情况就是，字符正好和图像缓冲区中的字对齐。在这种方式下，字符的位置限制为 32 个/行（256/8，假设 HRES=256），那么最多就可以显示 25 行文本（200/8，假设 VRES=200）。

更高级的方式需要给予每个字符在任何给定像素坐标处排列的绝对自由。这需要一种通常被称为 BitBLT（Bit Block Transfer，位块传输）的操作，该操作在计算机图形学中尤其在视频游戏设计中非常常见。在以下的内容中，我们将一直采用最简单的办法。我们将寻找一种解决办法，它只使用最少的硬件资源就可以完成字符打印工作。

13.21 通过视频打印文本

通过视频打印文本时，我们需要光标的支持，把它作为一个虚拟的占位符来跟踪屏幕上放置下一个字符的位置。在打印时，是很容易通过移动光标来模拟打字机以之字路线前进并滚动纸张的行为的。

> 在我写到这里时，我突然想到可能大部分读者都从来没有在现实生活中使用过打字机，那么这种类比的美感就完全没有了。也许大家感觉我就像在谈论古老的芦苇笔或羊皮纸……

光标由两个整数组成，分别存储新的坐标系的 *x* 坐标和 *y* 坐标，这个新坐标系和传统的笛卡尔坐标系正好相反，并且是用行和列而不是单个像素来作为坐标刻度的。

- ❑ cx，表示当前列，从左到右计数，范围为 0~31。
- ❑ cy，表示当前行，从上到下计数，范围为 0~24。

为了在屏幕上当前光标位置打印 ASCII 字符，我们要创建一个 putcV() 函数来执行以下几个步骤。

（1）检查所需字符是否在我们字体定义的范围内（从 ASCII 码 0x20 一直到 0x7F）：

```
void putcV( char a)
{
    int i, j, *p;
    const char *pf;

    // 1. check if char in range
    if ( a < F_OFFS)
         return;
    if ( a >= F_OFFS+F_SIZE)
         return;
```

（2）检查光标位置处于屏幕边界之内，在必要时进行换行和翻页：

```
// 2. check page boundaries and wrap or scroll as
necessary
if ( cx >= HRES/8)       // wrap around x
{
   cx = 0;
   cy++;
}
```

```
    if ( cy >= VRES/8)      // scroll up y
    {
    int *pd = VH;
    int *ps = pd+(HRES/32)*8;
    for( i=0; i<(HRES/32)*(VRES-8); i++)
      *pd++ = *ps++;
    for( i=0; i<(HRES/32)*8; i++)
      *pd++ = 0;
    // keep cursor within boundary
    cy=VRES/8-1;
  }
```

(3) 在图像缓冲区内查找对应于光标位置 (p) 的地址，并在 Font8x8[]数组 (pf) 中找到字符的定义：

```
// 3. set pointer to word in the video map
p = &VH[ cy * 8 * HRES/32 + cx/4];
// set pointer to first row of the character in font
array
pf = &Font8x8[ (a-F_OFFS) << 3];
```

(4) 逐字节地复制字符，注意在覆盖每个字符行时先清空背景色：

```
// 4. copy one by one each line of the character on
screen
for ( i=0; i<8; i++)
{
   j = (3-(cx & 3))<<3;   // consider MSB first
   *p &= ~(0xff << j);    // clear background
   *p |= ((*pf++) << j);  // overimposed character

   // point to next row
   p += HRES/32;
} // for
```

(5) 最后，移动光标位置：

```
    // 5. advance cursor position
    cx++;
} // putcV
```

把该函数加入到 graphic.c 文件末尾，并把它的原型加入到 graphic.h 头文件末尾：

```
void putcV( char a);
```

为了方便，现在创建一个小函数，在屏幕上打印整个 ASCII 字符串（以 0 结束）：

```
void putsV( char *s)
{
  while (*s)
    putcV( *s++);
  // advance to next line
  cx=0; cy++;} // putsV
} // putsV
```

把该函数加入到 graphic.c 库模块中，并把原型加入 graphic.h:

```
void putsV( char *s);
```

既然已经进行到这里了，那就索性再加入几个有用的宏到 graphic.h 文件中：

```
#define Home()          { cx=0; cy=0;}
#define Clrscr()        { clearScreen(); Home();}
#define AT( x, y)       { cx = (x); cy =(y);}
```

- Home()给出光标在屏幕左上角的位置。
- Clrscr()首先清除屏幕，然后重置鼠标位置到最顶端。
- AT(x,y)将光标置于期望的行（x）和列（y）处。

13.22 文本测试

为了快速测试新的文本函数的有效性，我们现在创建一个小程序，它在打印完屏幕第一行的一个小标题之后，再打印定义在 8×8 字体中的每个字符：

```
//*
** TextTest.c
**
*/
// configuration bit settings, Fcy=72MHz, Fpb=36MHz
#pragma config POSCMOD=XT, FNOSC=PRIPLL
#pragma config FPLLIDIV=DIV_2, FPLLMUL=MUL_18, FPLLODIV=DIV_1
#pragma config FPBDIV=DIV_2, FWDTEN=OFF, CP=OFF, BWP=OFF
#include <p32xxxx.h>
#include <explore.h>
#include <graphic.h>

main( void)
{
   int i;
  // initializations
  initEX16();  // init and enable vectored interrupts
  initVideo(); // start the state machines

  Clrscr();

  AT( 5, 2);
  putsV( "Exploring the PIC32!");

  AT( 0, 4);
  for( i=0; i<128; i++)
    putcV( i);
  while (1);

} // main
```

把这个文件保存为 TextTest.c，并将其加入到新的工程 TextTest 中。同时保证所有其他模块都已经加入到工程中，包括 graphic.c、graphic.h 和 explore.c。生成工程，对 Explorer 16 演示板采用在线调试器进行编程。如果一切顺利，就可以运行程序并在屏幕上看到正确的欢迎信息（如图 13-31 所示）。

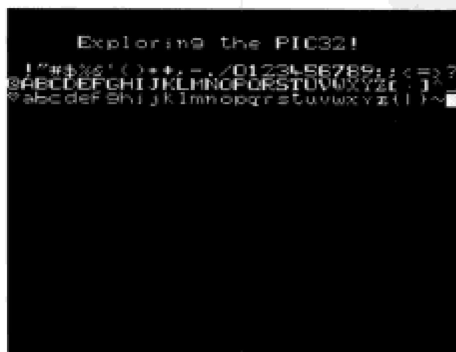


图 13-31 屏幕截图：文本测试

13.23 Matrix 程序的修改

为了进一步测试新的文本视频模块，我们现在修改一个在本书前面给出的示例：Matrix。那时候，我们使用异步串行通信模块（UART1）和 VT100 计算机终端进行通信，或者更确切一些，是和一个运行超级终端程序的 PC 机进行通信，超级终端配置为对 DEC VT100 终端协议进行仿真。现在把那时候用于传递字符到串行端口的函数调用 putcU() 替换成 putcV()，从而将字符直接传递到图像接口。

修改 TextTest 工程，把 TextTest.c 主模块替换成新的 Matrix2.c 模块，修改如下：

```
/*
** Matrix2.c
*/
// configuration bit settings, Fcy=72MHz, Fpb=36MHz
#pragma config POSCMOD=XT, FNOSC=PRIPLL
#pragma config FPLLIDIV=DIV_2, FPLLMUL=MUL_18, FPLLODIV=DIV_1
#pragma config FPBDIV=DIV_2, FWDTEN=OFF, CP=OFF, BWP=OFF
#include <p32xxxx.h>
#include <graphic.h>

#define COL     HRES/8
#define ROW     VRES/8

main()
{
  int v[ COL];    // vector containing length of each string
  int i,j,k;

  // 1. initializations
  initEX16();
  initVideo();
  Clrscr();         // clear the screen

  // 2. init each column length
  for( j =0; j < COL; j++)
          v[j] = rand()%ROW;

  // 3. main loop
  while( 1)
  {
     // 3.1 refresh the screen with random columns
     for( i=0; i<ROW; i++)
    {
      AT( 0, i);
      // refresh one row at a time
      for( j=0; j<COL; j++)
      {
        // fill random char down to each column length
        if ( i < v[j])
           putcV( '!'+(rand()%15));
        else
          putcV(' ');
        } // for j
    } // for i
```

```
        // 3.2 randomly increase or reduce each column length
        for( j=0; j<COL; j++)
        {
            switch ( rand()%3){
            case 0: // increase length
                v[j]++;
                if (v[j]>ROW)
                    v[j]=ROW;
               break;

        case 1:   // decrease length
              v[j]--;
              if (v[j]<1)
                  v[j]=1;
              break;

        default:// unchanged
             break;
        } // switch

      } // for j
    } // main loop
} // main
```

在保存并重新生成该工程之后，对 Explorer 16 演示板用在线调试器进行编程，并运行程序（如图 13-32 所示）。你将会看到屏幕更新的速度更快，那是因为现在直接访问视频存储器，信息传输没有串行连接的限制（和前一个 demo 工程中的连接波特率 115 200 一样快；这是一个瓶颈）。本程序运行得如此之快，以至于必须加入几毫秒的延迟才能让肉眼看清屏幕。

```
// 3.3 delay to slow down the screen update
Delayms( 5);
```



图 13-32 屏幕截图：Matrix 重载

13.24 小结

本章研究了使用最少的硬件资源（仅仅 3 个电阻）来产生视频信号输出的可能性。我们学

习了如何将 4 个外围设备模块组合起来建立产生 NTSC 复合视频信号所需的复杂机制。把一个 16 位的定时器、一个输出比较模块、一个 SPI 端口和一对 DMA 模块通道组合起来，就可以仅花费 1.5%的处理器开销来获取视频输出能力。在开发了基本的图形函数来画点和画线以后，我们介绍了图像视频输出可以提供的其他一些功能，包括一维和二维函数的曲线图绘制。最后简单介绍了如何画分形和用图形方式显示文本。

13.25 对 PIC24 行家的提示

PIC32 的 OC 模块和 PIC24 中的基本完全一样，不过仍然在设计中加入了一些重要的改进。在把 PIC24 程序移植到 PIC32 上时，以下所列几项内容会对代码产生影响。

（1）OCxCON 控制寄存器的布局有所改进，从而和其他大多数外围设备的控制寄存器布局更加接近，因此模块的 ON、FRZ 和 IDL 位现在可以更好地控制低功耗模式下的操作。

（2）加入了 OC32 控制位，在 OC 模块和 32 位定时器相连接时，可以使能 32 位操作模式。

13.26 提示与技巧

我们在图形世界中的简单旅行的最后一个亮点，就是给图形库加入一些动画能力。为了使动画流畅，并且避免屏幕上出现图像的干扰性锯齿，通常会使用一种被称为*双缓冲*（double buffering）的技术。这需要分配 2 个大小相同的图像缓冲区。一个是“活跃的”缓冲区，显示在屏幕上；另一个是“隐形”缓冲区，是真正进行绘画的地方。当隐形缓冲区中的绘画动作完成之后，两个缓冲区会交换。活跃缓冲区中的内容就再也看不见了。现在的隐形缓冲区可以被清空，而不用担心产生任何锯齿，绘画过程也可以重新开始。

在当前的图像分辨率设置（256×200）下，RAM 用量总数为 12 800 字节（256×200×2/8），这意味着仅仅使用了 PIC32MX360 单片机中可用 RAM 总量的 40%。

为了扩展我们的图形库并支持双缓冲，我们现在实施一些小改动。

- 在 graphic.c 模块头部，加入第二个图像缓冲区的声明：

```
#ifdef DOUBLE_BUFFER
int VMap2[ VRES*(HRES/32)]; // second image buffer
#endif
```

- 在 initVideo() 函数的 VA 和 VH 指针初始化的地方，加入一个新的条件赋值语句。(现在你能够理解我为什么使用 2 个指针来指向同一个图像缓冲区了。)

```
// 6. init the active and hidden screens pointers
VA = VMap1;
#ifdef DOUBLE_BUFFER
  VH = VMap2;
#else
  VH = VA;
#endif
```

- 加入一个新函数 ClearHScreen()，在双缓冲模式下清除隐形缓冲区的内容：

```
void clearHScreen( void)
{ // fill with zeros the Hidden Video array
  memset( VH, 0, VRES*( HRES/8));
  // reset text cursor position
  cx = cy = 0;
} //clearHScreen
```

- 加入 swapV() 函数来交换两个缓冲区（仅仅是两个指针的交换）：

```
void swapv( void)
{
  int * V;
  if ( VState == SV_LINE)   // wait end of the frame
    while ( VCount != 1);

  V = VA; VA = VH; VH = V;  // swap the pointers
  VPtr = VA;
} // swapV
```

需要注意，不能在一个图像帧的中间执行指针交换，但是可以和一帧的结束以及下一帧的开始保持同步。

加入最后一个函数，用于动画终止，显示模式回到单缓冲模式：

```
void singleV( void)
{ // make all functions work on a single image buffer
  VA = VMap1;
  VH = VA;
}
```

记住把以上所有函数的原型加入到 graphic.h 头文件中，并且附带在该文件头部加入新的标号 DOUBLE_BUFFER 的声明：

```
#define DOUBLE_BUFFER   // comment if single buffering required
void clearHScreen( void);
void swapV(void);
void singleV( void);
```



注解 本章以及之前章节中开发的所有示例程序，现在都可以使用新的扩展图像模块进行重新编译，只是需要将 DOUBLE_BUFFER 声明的注释去掉，或者在 initVideo() 调用之后立即调用 singleV() 函数！

13.27 练习

(1) 修改 Mandelbrot.c 文件，使用 32 位的定时器对 PIC32 的性能进行统计，并将时间和图像坐标显示在屏幕上。

(2) 创建一个组合 demo 程序，使用 PS/2 键盘输入和图像库文件来提供终端控制台的功能。

(3) 修改 graph2D.c 文件，允许用户通过 Explorer 16 演示板上的 4 个按钮来改变功能选项，包括增加和减少缩放比例、使用双缓冲动画技术在刷新屏幕时改变“顶点”的位置等。

(4) 进行 3D 几何函数的实验，画出物体的透视图并在三维空间中进行旋转。

13.28 参考书

Benoit B. Mandelbrot 所著的 *The Fractal Geometry of Nature*。这就是那本介绍分形的书。作者对分形理论的发现做出了巨大贡献。

Douglas Hofstadter 所著的 *Godel, Escher, Bach: An Eternal Golden Braid, 20th Anniversary Edition*。这是我的书架最让人激动的书之一，共有 777 页，非常难懂，但是它带我走上了通往图形、数学和音乐以及把这三者奇妙地连接在一起的旅程。

13.29 链接

http://en.wikipedia.org/wiki/Fractals。对分形知识进行在线了解的起点。

http://en.wikipedia.org/wiki/Zx_spectrum。Sinclair ZX Spectrum 是第一代个人计算机（过去通常被称为家庭计算机）之一，出现于 20 世纪 80 年代初。它的图形处理能力和本章中开发的图形库的处理能力很接近。尽管它使用了一些特制的逻辑部件来产生视频输出，它的处理能力还是比 PIC32 的处理能力低大概 10%。然而，它能够产生彩色图像的功能，尽管很有限（只有 16 种颜色，分辨率为 8×8 像素），仍然吸引着无数的程序员去创建富有挑战性和创造性的视频游戏。

第 14 章　大容量存储

14.1　计划

很多嵌入式控制应用都需要使用永久性数据存储空间，而其需求量又往往大于单片机本身提供的通用串行 EEPROM 和 Flash 程序存储器的容量。用户所需的存储容量可能是现有容量的成百上千倍，达几百兆字节甚至数吉字节。如果你有数码相机、MP3 播放器或者哪怕只有一个手机，就会理解消费类多媒体应用程序的存储容量需求，也会熟悉可用的海量存储技术。虽然硬盘驱动器变得越来越小，能耗也有所降低，但是市场上还是存在相当多的固态存储器（还是基于 Flash 技术的，比如 Compact Flash、Smart Media、Secure Digital、Memory Stick 等），而消费类应用对存储容量的大量需求，也导致固态存储器的价格已经压到很低，所以只要有可能，哪怕很麻烦也会把这些存储器集成到嵌入式控制器中去。

在本章中，我们将学习如何用最少的硬件资源把一种最常见和最廉价的大容量存储设备与 PIC32 单片机连接起来。

14.2　准备

除了 MPLAB IDE、MPLAB C32 编译器以及 MPLAB SIM 仿真器在内的这些常见软件工具之外，本章还需要用到 Explorer 16 演示板和用户自行选择的在线调试器。你还需要准备一个电烙铁和一些元器件，从而可以通过原型板区或者小扩展板来扩展 Explorer 16 演示板的功能。你还可以访问本书配套网站（www.exploringPIC32.com）来获取有关扩展板的更多信息，从而更好地完成本章中的实验。

14.3　探索

每一种有竞争力的大容量存储技术都有其优缺点，因为每一种技术的设计目标都针对不同的应用。我们将根据以下准则来挑选最适合我们应用的理想的大容量存储媒介：

- 是否具备存储空间和所需连接器；
- 物理接口（很可能是串行接口）所需的引脚数量；
- 存储容量；
- 提供开放的规范说明；
- 容易实现；
- 存储器和所需接头的价格。

Secure Digital（SD）卡的标准最满足以上所有的要求，现如今它是数码相机和很多其他多媒体消费类设备最常采用的一种大容量存储媒介。从 SD 卡的规范中可以看出以前称为多媒体卡（Multi Media Card，MMC）的技术的演变，至今这两种卡在电气特性和机械特性上还保持了部分（向后）兼容。安全数字卡协会（SDCA）掌握和控制着 SD 存储卡的技术规范标准，他们要求所有计划积极参与设计、开发、制造或销售使用 SD 规格的产品的公司，成为该协会成员。在写这本书时，一般的 SDCA 会员需要缴纳 2000 美元的年费。相反，多媒体卡协会（MMCA）并不要求使用者成为会员，但是要想获得 MMC 规格的副本则必须付钱，起价 500 美元。因此两种技术无论从哪方面来说都离免费或者说“开放”还很远。

幸运的是，SDCA 面向公众发布了“简化的物理规范”，它是 SD 规范的一个“子集”。这些信息足够我们用来开发一个能体现 SD/MMC 存储技术的基本原型，并开始设计 PIC32 的大规模存储接口。

14.4 物理接口

SD 卡仅需要 9 个电气触头和一个 SD/MMC 兼容的连接器，只花几美元就能买到。连接器仅需要再使用 2 个引脚，用来进行插入检测和写保护开关状态感知。有两种主要的通信模式可以使用：第一种（称为 SD 总线）源于 SD/MMC 标准，需要一个 4 位宽的总线接口；第二种模式是串行的，基于流行的 SPI 总线标准。是第二种模式使 SD/MMC 大规模存储设备格外受到所有嵌入式控制应用的欢迎，因为大部分单片机要么配备了硬件 SPI 接口，要么能够很容易地用很少的 I/O 引脚来模拟一个 SPI 接口（bit-banging，位脉冲）。最后，SD/MMC 卡的物理规范表明，对于所有采用先进 CMOS 工艺实现的现代单片机的嵌入式应用来说，2.0V 到 3.6V 的工作电压是最理想的。PIC32MX 系列正是属于这种情况（如图 14-1 所示）。

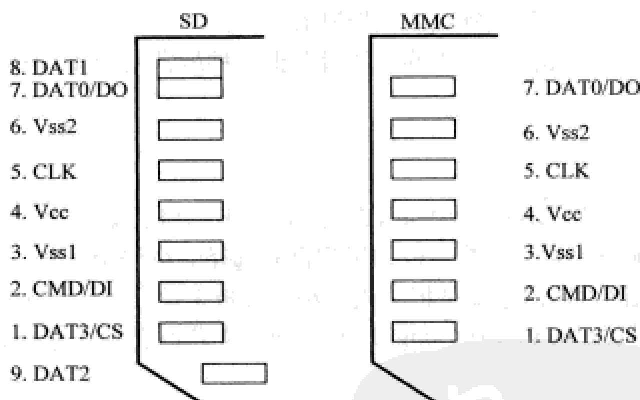


图 14-1　SD 卡和 MMC 卡的连接器引脚分布

注解　miniSD 卡、microSD 卡和 SD 卡在逻辑和电气特性上是一样的。只是在形状、体积和引脚数量上和原始的标准有所差异。miniSD 卡和 microSD 卡的设计目的都是为了满足特殊的体积需求。只要配备一个适配器或者合适的连接器，它们就能用在以下应用中。

14.5 和 Explorer 16 演示板连接

尽管 SPI 接口所需的电气连接引脚数量很少，但是市场上提供的所有 SD/MMC 卡的连接器都是仅面向表面贴装应用设计的，因此几乎不可能用实验电路板的方式或者利用 Explorer 16 演示板的原型区来连接存储卡。

在前一章，我们使用了第一个 SPI 外围设备模块（SPI1）来产生视频输出，然而该程序无法和其他程序共享该资源，因此我们使用第二个 SPI 模块（SPI2），让 SD 卡接口和 EEPROM 接口通过单独的片选信号（CS）来共享该模块。除了常用的 SCK、SDI 和 SDO 引脚，我们还为 SD/MMC 连接器中不使用的引脚（保留给 4 位宽的 SD 总线接口）以及另外两个将用作存储卡探测信号和写保护信号的两个引脚提供上拉电阻（如图 14-2 所示）。

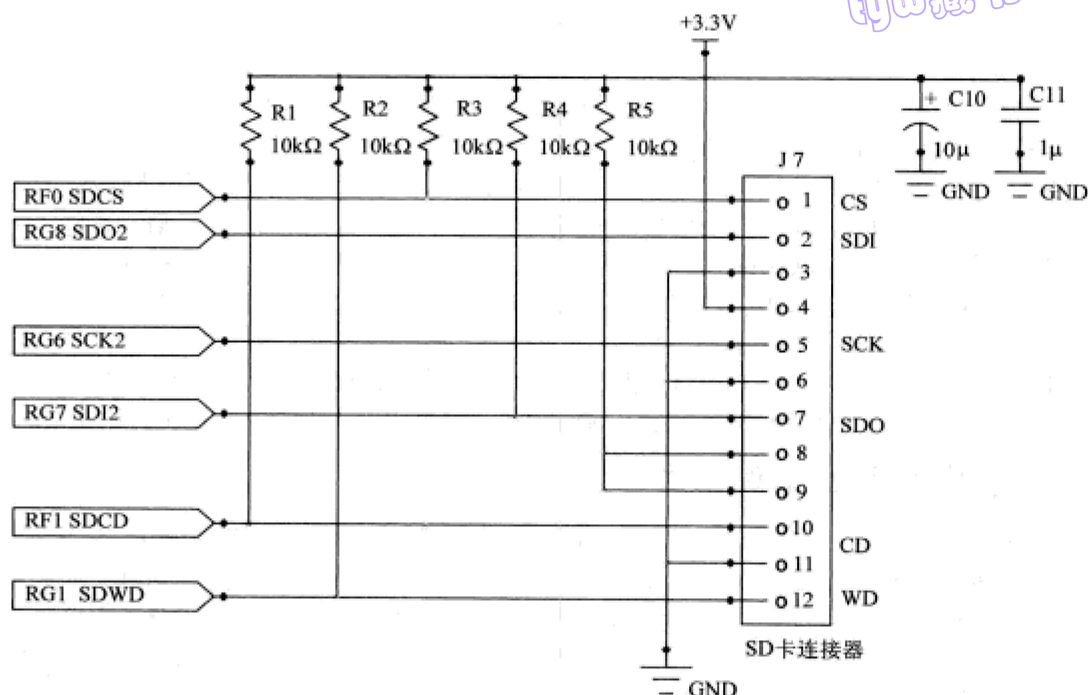


图 14-2 SD/MMC 存储卡和 Explorer 16 演示板的接口

> **注解** Microchip 公司最近针对 SD 卡和 MMC 卡（AC164122）开发了称为 PICTail 的扩展板，它能有效地用于本章列出的所有工程。本书配套网站（www.ExploringPIC32.com）上提供了支持新的 PICTail 扩展板的另一套可选的引脚分布方案。

14.6 开始一个新工程

创建一个新工程，将其命名为 SDMMC，然后编写基本的初始化例程，对所有必要的 I/O 引脚以及 SPI2 模块的配置进行初始化。

```
/*
** SDMMC.c SD card interface
*/
#include <p32xxxx.h>
#include <sdmmc.h>

// I/O definitions
#define SDWP    _RG1    // Write Protect input
#define SDCD    _RF1    // Card Detect input
#define SDCS    _RF0    // Card Select output

void initSD( void)
{
    SDCS = 1;           // initially keep the SD card disabled
    _TRISF0 = 0;        // make Card select an output pin

    // init the SPI2 module for a slow (safe) clock speed first
    SPI2CON = 0x8120;   // ON, CKE=1; CKP=0, sample middle
```

```
  SPI2BRG = 71;// clock = Fpb/144 = 250kHz
} // initSD
```

特别说明一下，我们需要在 SPI2CON 寄存器中将 SPI 模块配置成工作在主模式下，同时还需要设置时钟信号的极性、时钟沿、输入采样点以及初始时钟频率。时钟输出信号（SCK）必须使能，并且在空闲状态下保持为低。SDI 输入的采样点必须居中。频率由 SPI 波特率产生器（SPI2BRG）控制，SPI2BRG 将对外围设备时钟信号（T_{pb}）分频。上电以后一直到 SD 卡被正确地初始化以前，必须保持 SPI 时钟速率安全地设置在 400kHz 以下；因此我们将其设置为 $T_{pb}/144$，从而获得一个 250kHz 的时钟信号。不过这也只是一个临时设置；在发送了前面几个命令之后，就可以大大加快通信速度。

注意只有 SDCS 信号（RF0 引脚）需要手动配置为输出引脚，而 SCK2 和 SDO2（对应于 RG6 和 RG8 引脚）在 SPI2 模块使能以后，就会立刻自动配置为输出引脚。

14.7 选择 SPI 的操作模式

当 SD/MMC 卡插入到连接器并上电以后，它就开始工作在默认通信模式 SD 总线模式下。为了告知存储卡我们希望采用 SPI 模式进行通信，就必须选择存储卡（SDCS 引脚变低）并开始发送第一个复位命令。一旦存储卡进入 SPI 模式以后，除非重新启动，否则就不能再变回到 SD 总线模式。但是，这也意味着如果存储卡没有经过确认就从连接口中拔出，然后又插入，那就必须确保初始化例程或者至少复位命令会被重复执行一次，从而回到 SPI 模式下。可以在任何时候通过检查 SDCD 信号线（RF1 输入引脚）的状态来检测存储卡的状态。

14.8 在 SPI 模式下发送命令

在 SPI 模式下，命令是通过 6B 的封装包形式发送给 SD/MMC 卡的，所有来自于 SD 卡的响应则是不同长度的多字节数据块。于是，只需要采用常见的基本 SPI 例程和存储卡进行通信，每次发送或接收一个字节即可（从前一章里已经得知，发送和接收操作其实是一样的）。

```
// send one byte of data and receive one back at the same time
unsigned char writeSPI( unsigned char b)
{
  SPI2BUF=b;                       // write to buffer for TX
  while( !SPI2STATbits.SPIRBF);    // wait transfer complete
  return SPI2BUF;                  // read the received value
}// writeSPI
```

为了提高代码的可读性和易用性，我们还定义了另外两个宏，将同一个 writeSPI() 函数定义为 readSPI()，或者定义为时钟输出函数 clockSPI()。两个宏都将发送一字节的虚拟数据（0xFF）。

```
#define readSPI()   writeSPI( 0xFF)
#define clockSPI()  writeSPI( 0xFF)
```

为了发送命令，首先选择存储卡（将 SDCS 置低），并通过 SPI 端口发送一个数据包，该数据包包含以下 3 部分内容。

- 第一部分是 1B，包含一个命令索引。以下给出的这些定义包含了在本工程中将要用到的所有命令。

```
// SD card commands
#define RESET          0    // a.k.a. GO_IDLE (CMD0)
#define INIT           1    // a.k.a. SEND_OP_COND (CMD1)
```

```
#define READ_SINGLE     17
#define WRITE_SINGLE    24
```

- 命令索引之后是一个 32 位的存储器地址。它是一个无符号整数（32 位），必须先传输高位。
- 最后是 1B 的 CRC，作为命令包的结束。

SD 总线模式中始终使用循环冗余校验（CRC），以确保总线上传输的每一个命令和每一个数据块都是正确的。但是，一旦发送复位命令并切换到 SPI 模式下，就会自动禁止 CRC 保护，CRC 值也会被忽略。实际上，从那时候开始，就已经假设存储卡和主机之间（本例中即为 PIC32）已经建立起了直接而可靠的连接。利用这个默认行为，可以通过一个预先计算好的值来简化代码。这个值即为 RESET 命令的 CRC 码。对于所有后续命令，CRC 域都被认为是“无需关注的”。以下给出的是 sendSDCmd() 函数的第一部分，我们将利用这个函数向 SD 卡发送各种命令。

```
int sendSDCmd( unsigned char c, unsigned a)
// c command code
// a byte address of data block
{
    int i, r;

    // enable SD card
    enableSD();
// send a comand packet  (6 bytes)
writeSPI( c | 0x40);      // send command
writeSPI( a>>24);         // msb of the address
writeSPI( a>>16);
writeSPI( a>>8);
writeSPI( a);             // lsb

writeSPI( 0x95);          // send CMD0 CRC
```

把所有 6B 都发送到 SD 卡以后，就该等待接收响应信息了。事实上，继续持续发送虚拟数据到 SPI 端口是很重要的。响应数据应该是 0xFF；SDI 信号线将会保持为高，一直到存储卡准备好发送正确的响应码。存储卡规范中说明，在收到正确响应之前，最多需要 64 个时钟周期或者说 8B 的传输时间。如果超过了这个时间限制，就必须认为存储卡发生了重大故障，并取消通信。

```
    // now wait for a response, allow for up to 8 bytes delay
    for( i=0; i<8; i++)
    {
      r=readSPI();
      if ( r != 0xFF)
      break;
    }
    return ( r);

    // NOTE CSCD is still low!
} // sendSDCmd
```

如果接收到了响应码，那么响应码中置位的位会指出发生的问题是什么（如表 14-1 所示）。

表 14-1　SD 卡的命令响应码

位	描　述
0	空闲状态
1	擦除复位
2	非法命令
3	通信 CRC 错误
4	擦除顺序错误
5	地址错误
6	参数错误
7	0（一直保持）

需要注意在 sendSDCmd() 函数返回时，必须仍然使 SD 卡保持为选择状态（SDCS 为低），从而让那些需要发送或从存储卡接收额外数据的命令（比如块写入命令和块读取命令）能够继续执行。对于所有其他不需要继续进行额外数据传输的命

令，则必须在函数调用之后立即取消对存储卡的选择（将 SDCS 置为高）。另外，因为我们希望将 SPI2 端口和其他外围设备（比如 Explorer16 演示板上加载的串行 EEPROM）进行共享，所以必须保证 SD/MMC 卡在片选信号（SDCS）上升沿之后，能够再接收几个时钟周期的数据（8 个周期就足够了）。根据 SD/MMC 规范可知，这可以使存储卡完成几个重要的自身维护任务，包括正确释放 SDO 信号线，这对于总线上的其他设备能正确通信是非常必要的。

以下是另外一组宏，能够持续执行该任务：

```
#define disableSD() SDCS = 1; clockSPI()
#define enableSD() SDCS = 0
```

14.9　完成 SD 卡的初始化

在存储卡能够有效地用于大规模存储应用之前，必须完成一系列定义好的命令。这个命令序列在原始的 MMC 卡规范中已经定义，在 SD 卡规范中则进行了一些细微的改动。因为我们并不打算使用专门面向 SD 卡标准的任何高级特性，所以只采用针对 MMC 卡定义的基本命令序列以获取最大限度的兼容性。在这个命令序列中包含 5 个步骤，存储卡一插进连接器并上电以后，这个序列就开始了。

（1）CS 信号线初始化为高电平（没有选择存储卡）。

（2）在存储卡可以接收命令之前，必须提供超过 74 个时钟脉冲。

（3）存储卡必须被选择。

（4）发送 RESET（CMD0）命令；存储卡必须以进入到空闲状态（并激活 SPI 模式）来进行响应。

（5）重复发送 INIT（CMD1）命令，直到存储卡退出空闲状态。

以下给出的是函数 initMedia()的代码段，这段代码就是完成以上给出的 5 个步骤的。

```
int initMedia( void)
// returns 0 if successful
//          E_COMMAND_ACK   failed to acknowledge reset command
//          E_INIT_TIMEOUT  failed to initialize
{
  int i, r;

  // 1. with the card NOT selected
  disableSD();

  // 2. send 80 clock cycles start up
  for ( i=0; i<10; i++)
    clockSPI();

  // 3. now select the card
  enableSD();

  // 4. send a single RESET command
  r = sendSDCmd( RESET, 0); disableSD();
  if ( r != 1)              // must return Idle
  return E_COMMAND_ACK;     // comand rejected

  // 5. send repeatedly INIT until Idle terminates
  for (i=0; i<I_TIMEOUT; i++)
  {
    r = sendSDCmd( INIT, 0); disableSD();
    if ( !r)
```

```
        break;
    }
    if ( i == RI_TIMEOUT)
      return E_INIT_TIMEOUT; // init timed out
```

初始化命令的完成需要一定的时间，具体取决于存储卡的大小和类型，通常为几十分之一秒。因为我们工作在 250kbit/s 下，每个字节的发送需要 32μs。如果考虑到每个命令有 6B 的重发行为，那么根据 SD 卡规范可知，设置最大值为 10 000 的计数器，超时限制（I_TIMEOUT）的值大约是三十分之一秒。只有在成功完成上述命令序列以后，才能最终改变操作方式，将时钟速率提高到硬件能支持的最高值。做一点小实验就能够发现，带有提供 SD/MMC 连接器子板的 Explorer 16 演示板，可以很容易地将时钟频率提高到 18MHz。重新配置 SPI 的波特率产生器比率为 1:2 就可以做到这一点。现在可以加入下列代码段来完成 initMedia() 函数：

```
    // 6. increase speed
    SPI2CON = 0;        // disable the SPI2 module
    SPI2BRG = 0;        // Fpb/(2*(0+1))= 36/2 = 18MHz
    SPI2CON = 0x8120;   // re-enable the SPI2 module
    return 0;

  } // init media
```

14.10 从 SD/MMC 卡读取数据

SD/MMC 卡是典型的包含大型 Flash 存储阵列的固态设备，因此我们可以期待能够在任何地址处读取或写入任意长度的数据（在卡容量范围内）。而在实际应用中，考虑到和之前传统的大容量存储技术的兼容性，在进行存储器访问时还是存在一些限制。实际上，所有的操作都是以固定大小的数据块为单位来进行的，默认大小为 512B。512B 正好是典型的个人计算机硬盘中数据“扇区”的标准大小，这并非巧合。尽管通过命令可以改变数据块的大小，但是我们仍然维持默认设置，从而能在后续实验中利用这个兼容性。在下一章里，我们将开发一组例程来实现一个完整的和大部分通用 PC 操作系统兼容的文件系统。这样一来，我们就能够访问通过 PC 写入 SD 卡的文件，同时 PC 也可以访问通过我们的设备写入到 SD 卡上的文件。

使用 READ_SINGLE（CMD17）命令，就可以发起一次在给定存储器地址处的单个扇区数据的传输。该命令的参数是一个 32 位的“字节地址”，但是在访问多个扇区数据时，就会经常用到 LBA（Logical Block Address，逻辑块地址），这个术语是从其他大容量存储应用领域中借过来的。

```
Typedef unsigned LBA;    //logic block address, 32 bit wide
```

为了避免混淆，在以下内容中我们将统一使用 LBA 或块地址，并且在把参数传递给 READ_SINGLE 命令之前，会把 LBA 值乘以 512 来得到实际的字节地址。

向 SD 卡写入一个扇区的数据需要以下 5 个步骤。

（1）发送一个 READ_SINGLE 命令。

（2）等待 SD 卡给出带有特殊令牌 DATA_START 的响应码。这是存储卡用以告诉用户已准备好数据块发送的方式。

因为存储卡可能需要一点时间来定位数据块，就像在初始化阶段一样，所以设置一个超时限制是很重要的。在等待数据令牌时，只有 readSPI() 函数被重复调用，该函数每次发送/接收 1B（频率为 18MHz），那么 25 000 的超时计数器（R_TIMEOUT）就可以提供不超过 1ms 的有效时间限制。

（3）一旦接收到 DATA_START 令牌，就可以快速顺序读取组成所请求数据块的 512B 了。

（4）数据块之后是一个必须读取的 16 位 CRC 值，但是在其他情况下则可以丢弃。也正是在这时候我们应该取消存储卡的选择并终止整个读取命令序列。

例程 readSECTOR() 用以下几行代码来执行整个读取过程：

```
#define DATA_START    0xFE

int readSECTOR( LBA a, char *p)
// a        LBA of sector requested
// p        pointer to sector buffer
// returns  TRUE if successful
{
  int r, i;

  // 1. send READ command
  r = sendSDCmd( READ_SINGLE, ( a << 9));
  if ( r == 0)    // check if command was accepted
  {

    // 2. wait for a response
    for( i=0; i<R_TIMEOUT; i++)
    {
      r = readSPI();
      if ( r == DATA_START)
        break;
    }

    // 3. if it did not timeout, read 512 byte of data
    if ( i != R_TIMEOUT)
    {
      i = 512;
      do{
        *p++ = readSPI();
      } while (--i>0);

      // 4. ignore CRC
      readSPI();
      readSPI();

    } // data arrived
  } // command accepted

  // 5. remember to disable the card
  disableSD();

  return ( r == DATA_START);    // return TRUE if successful
} // readSECTOR
```

注解　为了形象地说明存储卡的行为与硬盘驱动器以及软盘驱动器行为的相似性，我们可以将 Explorer 16 演示板上的一个 LED 灯设置为“读取”LED，通过观察 LED 的状态来防止用户在存储卡正在使用时将其移走。LED 灯可以在每次读取命令之前点亮而在结束时熄灭。

当然也可以采用其他方式。比如，和通常见到的 USB Flash 驱动器类似，存储卡一旦初始化就让 LED 灯变亮，而不管是否正在执行命令。只有在调用去初始化例程以后，才将 LED 灯熄灭，从而指示此时用户可以将卡移除。

14.11 向 SD/MMC 卡写入数据

基于和编写 readSECTOR() 函数时一样的考虑，我们在编写 writeSECTOR() 函数时，也将操作限制为针对 512B 的数据块。写入序列和你期望的一致，以 WRITE_SINGLE 命令开始，一共包含 5 个步骤。但这次的数据传输方向是相反的。

(1) 发送一个 WRITE_SINGLE 命令并检查 SD 卡的响应，从而确保命令被成功接收。

(2) 发送 DATA_START 令牌，然后立即以循环的方式发送所有 512B 的数据。

(3) 发送 2B 的虚拟数据充当 16 位的 CRC 值，因为在 SPI 模式下 CRC 检查是禁止的。

(4) 检查 SD 卡的响应。如果收到 DATA_ACCEPT 令牌，就可以判定整个数据块已经接收，写操作已经开始。

(5) 等待写命令的完成。当存储卡正在写入时，SDO 信号线保持为低。因此将等待 SDO 信号线重新变高。还是要设置一个超时值，限制存储卡完成操作所允许花费的时间。因为所有的 SD/MMC 存储器都是基于 Flash 存储技术的，因此可以推断写操作所需时间会通常比读操作所需时间要长得多。250 000 的超时值（W_TIMEOUT）将提供 100ms 的时间限制，对于市场上提供的最慢的存储卡来说，写入时间也足够了。

上述 5 个步骤完成之后，需要取消对存储卡的选择并终止整个写命令序列。

```
#define DATA_ACCEPT              0x05

int writeSECTOR( LBA a, char *p)
// a        LBA of sector requested
// p        pointer to sector buffer
// returns  TRUE if successful
{

  unsigned r, i;

  // 1. send WRITE command
  r = sendSDCmd( WRITE_SINGLE, ( a << 9));
  if ( r == 0)     // check if command was accepted
  {

    // 2. send data
    writeSPI( DATA_START);

    // send 512 bytes of data
    for( i=0; i<512; i++)
      writeSPI( *p++);

    // 3. send dummy CRC
    clockSPI();
    clockSPI();

    // 4. check if data accepted
    r = readSPI();
    if ( (r & 0xf) == DATA_ACCEPT)
    {

      // 5. wait for write completion
      for( i=0; i<W_TIMEOUT; i++)
      {
        r = readSPI();
```

```
        if ( r != 0 )
          break;
      }
    } // accepted
    else
      r = FAIL;

  } // command accepted

  // 6. remember to disable the card
  disableSD();

  return ( r);      // return TRUE if successful

} // writeSECTOR
```

> **注解** 和 readSECTOR()函数相似，可以用第二盏 LED 灯来表示写操作的执行并通知给用户。如果在写入过程中存储卡被移除，数据很有可能丢失或损坏。

把目前所写的源代码保存在一个文件中，并命名为 SDMMC.c，放置于 lib 目录下。在接下来的几章中我们会经常用到这个文件。

最后一件事是将以下两个函数加入进去，用来管理 SD/MMC 连接器的开关。

```
// SD card connector presence detection switch
int getCD( void)
// returns  TRUE card present
//          FALSE card not present
{
  return !SDCD;
}
```

当存储卡插进连接器以后，卡探测开关会关闭，SDCD 输入引脚被拉低。getCD()函数用于检测存储卡的生存性，如果返回 TRUE 则表示存储卡存在并已准备使用。

类似地，如果存储卡上的写保护开关并未处于“锁定”位置，而存储卡正处于插入状态，那么写保护开关将关闭，相应的 SDWP 输入引脚被拉低。

```
// card Write Protect tab detection switch
int getWP( void)
// returns  TRUE write protect tab on LOCK
//          FALSE write protection tab OPEN
{
  return SDWP;
}
```

getWP()函数在存储卡被正确插入时调用，如果存储卡处于锁定状态则返回 TRUE。

注意 SD/MMC 卡上的写保护开关和磁带以及 VHS 录音带上的保护开关类似，用来表示设备被锁定从而不能写入数据。

因此我们应该尊重用户的意愿，在 writeSECTOR()函数的最开始检查写保护开关，如果发现处于锁定状态则立刻终止写操作。

```
// 0. check Write Protect
  if ( getWP())
    return FAIL;
```

最后，创建一个新的头文件 SDMMC.h，保存在通用 include 目录下，文件中包含 SD/MMC 接口模块中所用到的函数原型和一些基本定义：

```
/*
** SDMMC.h SD card interface
*/
#define FAIL     FALSE
// Init ERROR code definitions
#define E_COMMAND_ACK      0x80
#define E_INIT_TIMEOUT     0x81

typedef unsigned LBA;      // logic block address, 32 bit wide

void initSD( void);        // initializes I/O pins and SPI
int initMedia( void);      // initializes the SD/MMC memory device
int getCD();               // chech card presence
int getWP();               // check write protection tab
int readSECTOR ( LBA, char *); // reads a block of data
int writeSECTOR( LBA, char *); // writes a block of data
```

14.12 测试 SD/MMC 接口

不管你是否相信，刚才开发的 4 个小字体的例程就是我们用来访问 SD/MMC 存储卡提供的看起来无穷大的“存储空间”的。例如，一个 1GB 的 SD 卡会提供大约 2 000 000（对，就是 200 万）个独立寻址的存储器块（扇区），每个块为 512B。而当前，这种容量的 SD/MMC 卡在美国市场上的零售价格通常还不到 20 美元！

我们来开发一个小的测试程序，说明 SD/MMC 模块的使用方法。该程序是模拟一个比较典型的应用，该应用需要把大量的数据保存到 SD/MMC 存储卡上。先把固定数量的数据块写入到预先确定的地址范围内，然后再读出来，验证整个过程是否成功。我们将使用 LCD 来报告诊断信息并跟踪整个过程。

创建一个新的源文件，名称为 RWTest.c，加入常见的说明以及与处理器相关的头文件，然后加入新的 sdmmc.h 头文件：

```
/*
**    RWTest.c
**
*/
// configuration bit settings, Fcy=72MHz, Fpb=36MHz
#pragma config POSCMOD=XT, FNOSC=PRIPLL
#pragma config FPLLIDIV=DIV_2, FPLLMUL=MUL_18, FPLLODIV=DIV_1
#pragma config FPBDIV=DIV_2, FWDTEN=OFF, CP=OFF, BWP=OFF

#include <p32xxxx.h>
#include <explore.h>
#include <LCD.h>
#include <SDMMC.h>
```

定义两个字节数组，每个数组的长度是默认的 SD/MMC 存储块的大小，即 512B：

```
#define B_SIZE      512    // data block size
char data[ B_SIZE];
char buffer[ B_SIZE];
```

测试程序会给第一个数组填充特定的、易于识别的数据，并将其内容重复写入到存储卡上。用 2 个常数来定义选择好的地址范围：

```
#define START_ADDRESS   10000   // start block address
#define N_BLOCKS        10      // number of blocks
```

连接在 Explorer 16 演示板的 PORTA RA2 引脚上的 LED2，给用户反馈 SD 卡的当前使用状态。需要注意的是，哪怕使用的是 PIC32 Starter Kit，这个 I/O 引脚也是可用的，因此 JTAG 端口是可用的。

```
#define LED    _RA2
```

现在开始写主程序，前面几行对 SD/MMC 模块在 LCD 上的输入输出内容进行初始化。

```
main( void)
{
  LBA addr;
  int i, j, r;

  // 1. initializations
  initEX16();
  initLCD();                // init LCD module
  initSD();                 // init SD/MMC module

  // 2. fill the buffer with pattern
  for( i=0; i<B_SIZE; i++)
    data[i]= i;
```

接下来的代码段提示用户插入存储卡，并用一个循环来检查 SD 卡是否有效。为了消除弹跳效应，这里设置了一小段延迟，延迟之后开始执行初始化例程，让存储卡准备好接收 SPI 命令。

```
// 3. wait for the card to be inserted
putsLCD( "Insert card..");
while( !getCD());         // check CD switch
Delayms( 100);            // wait contacts de-bounce
if ( initMedia())         // init card
{ // if error code returned
  clrLCD();
  putsLCD( "Failed Init");
  goto End;
}
```

准备好之后，进入实际的写数据阶段。LED 灯变亮，指示 SD 卡正在使用，并且在 LCD 显示器的第一行显示状态信息。两个嵌套循环重复调用 writeSECTOR() 函数，将开始于绝对 LBA=10,000 的 16 组数据写入到存储卡上，每组包含 10 个扇区。每写入 10 个扇区（大概 5KB），就在 LCD 显示屏的第二行上加入一个砖形字符（小黑块），从而形成一个进度条。如果任何一个写入命令失败，整个过程就会立刻停止，并在 LCD 上显示错误信息。

```
// 4. fill 16 groups of N_BLOCK sectors with data
LED = 1;                  // SD card in use

clrLCD();
putsLCD( "Writing\n");
addr = START_ADDRESS;
for( j=0; j<16; j++)
{
  for( i=0; i<N_BLOCKS; i++)
  {
    if (!writeSECTOR( addr+i*j, data))
    { // writing failed
```

```
            putsLCD( "Failed to Write");
            goto End;
        }
    } // i
    putLCD( 0xff);
} // j
```

现在将每个扇区的数据读取出来并验证其内容是否正确。在 LCD 指示新的阶段开始之后，用同样的两个嵌套循环来执行每组 10 个扇区的读取和验证过程。在读取并验证完一组扇区之后，新的砖形字符（黑色进度条）会显示在屏幕上用来指示进度。如果任何一步出错，整个过程都会立刻停止，并在 LCD 上显示错误信息。

```
// 5. verify the contents of each sector written
clrLCD();
putsLCD( "Verifying\n");
addr = START_ADDRESS;
for( j=0; j<16; j++)
{
  for( i=0; i<N_BLOCKS; i++)
  { // read back one block at a time
    if (!readSECTOR( addr+i*j, buffer))
    { // reading failed
      putsLCD( "Failed to Read");
      goto End;
    }

    // verify each block content
    if ( memcmp( data, buffer, B_SIZE))
    { // mismatch
      putsLCD( "Failed to Match");
      goto End;
    }
  } // i
  putLCD(0xff);
} // j
```

这里使用了标准 C 的 string.h 库中的 memcmp() 函数来有效地执行数据比较。如果两个缓冲区中的内容相同则返回 0，否则返回一个非零值。

如果一切运行正确，就会在 LCD 上打印出成功信息，并且因为 SD 卡不再使用，能够从连接器上移除，所以 LED 灯熄灭。

```
  // 7. indicate successful execution
  clrLCD();
  putsLCD( " Success!");

End:
  LED = 0;    // SD card not in use
  // main loop
  while( 1);

} // main
```

确认已经把所有需要的源文件（SDMMC.h、SDMMC.c、LCDlib.c、explore.c 和 RWTest.c）加入到工程中，生成工程，并用在线调试器对 Explorer 16 演示板进行编程。本次测试的执行需要一个在本章一开始就介绍过的带有 SD/MMC 连接器的子板以及一个空的 SD 卡。



注意　这是一件真事儿！当你运行 RWTest 程序时，SD 卡的内容将被修改，卡上的原有数据会被覆盖，因此所有的文件都会损坏。确认你已经把所有的家庭照片和钟爱的 MP3 文件保存在别的什么地方了！在下一章中，我们才会开发一个和通用 PC“文件系统”兼容的库文件。这个库文件开发出来以后，我们会使用通用格式来读写数据，从而不必担心损坏现有文件。

在运行代码时，看到 PIC32 在几秒种之内就能正确地执行完整个程序，这种喜悦之情会大大补偿构建 SD/MMC 接口的艰辛（还包括购买存储卡的代价）。

同时可以看到，我们使用的代码体积和资源是多么地少啊（见图 14-3）！

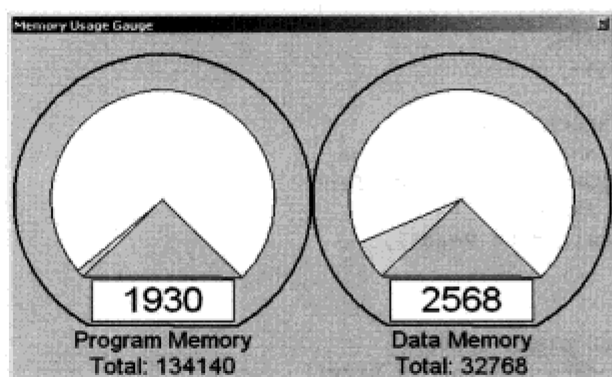


图 14-3　MPLAB 存储使用计量

总的来说，测试程序和 SD/MMC 访问库模块只用到了处理器 Flash 程序存储器中的 1 930 个字，也就是说小于可用存储器总量的 2%。更何况，就和前面的课程中一样，没有依靠任何编译器优化就获得了这样的结果。

14.13　小结

依我个人的意见，无论采用何种大容量存储技术，本方案都是最便宜和最简单的。毕竟，我们只是用了一组上拉电阻、一个便宜的连接器和几个 I/O 引脚就大大扩展了应用程序的存储能力。而在 PIC32 所需的资源方面，则只使用了 SPI 外围设备模块，并且还可以和其他应用共享使用。

本方法虽然简单，但也有其明显的缺陷。数据只能按照固定大小的块来写入，而且它在存储阵列里的位置是完全由具体的程序来决定的。换句话说，没有办法和个人计算机或者其他可以访问 SD/MMC 存储卡的设备共享数据，除非开发的是一个“定制”程序。更可怕的是，如果试图使用已经被 PC 机使用过的存储卡，那么 PC 数据很可能已损坏，整个卡需要完全重新格式化。在下一章里，我们将开发一个完整的文件系统库来着重讲解这个问题。

14.14　提示与技巧

选择以默认大小为 512B 的数据块来进行操作基本上是基于一些历史原因。本章开发的底层访问例程，都和大部分其他大容量存储设备（包括硬盘驱动器）的标准大小相符合，这样我

们就可以更容易地开发下一层应用（文件系统）。但是如果要寻求最好的性能，这可能不是正确的选择。实际上，如果追求更快写入速度，最好是使用更大的数据块；写入速度通常也是每一种 Flash 存储介质的瓶颈所在。

Flash 存储器通常提供很快的数据访问速度（读取），但是写入速度却相对较慢。写入需要两个步骤：首先，必须擦除一大块数据（通常被称为一个页）；然后才能在小一些的存储块上进行实际的写入操作。存储阵列越大，擦除页也会相对越大。例如，在一个 512MB 的存储卡上，擦除页很容易就超过 2KB 了。尽管这些细节对用户来说通常是隐藏的，因为存储卡内部的主控制器会负责擦除/写入的排列和缓冲，但是对应用程序的整体性能还是有一定的影响。如果假设一个特定的 SD 卡的页大小为 2KB，那么写入任何大小的数据（小于 2K）都需要内部控制器执行以下步骤。

- 把整个 2KB 的数据块内容读取到内部缓冲区中。
- 擦除，并等待擦除完成。
- 用新的数据替换缓冲区中某一部分的内容。
- 将整个 2KB 数据写回，并等待写回完成。

因为每次操作的存储卡大小都是 512MB，那么 2KB 的数据写入需要 SD 卡控制器把上述过程完整地执行 4 次，而通过改变数据块的大小或者使用一个多块写入命令，就可以只执行一次。在本例中，采用这些方法理论上可以将写入速度提高 400%，但因为代价会相当高，因此需要谨慎采用。实际上需要考虑以下一些不利因素。

- 生产厂商可能并不知道或者保证实际的存储页大小是多少，尽管可以确定增加 Flash 存储介质的密度（因此也增大了页大小）是非常安全的。
- 在 PIC32 应用程序中分配的 RAM 缓冲区的大小会增加，而 RAM 在任何嵌入式应用中都是非常宝贵的资源。
- 如果数据块大小改变，那么可能会增大更高软件层的开发难度（在下一章中将进行讲述）。
- 缓冲区越大，那么如果在缓冲区充满之前就把卡移走，数据丢失的可能性也越大。

14.15 练习

（1）采用不同的数据块大小进行实验，看看能给 SD 卡提供最佳写入性能的数据块大小是多少。这样就能间接地提示你得知存储卡生产厂商在 Flash 存储设备中使用的实际页大小是多少。

（2）采用多块写入命令或者改变数据块长度的方法来进行实验，以验证 SD 卡控制器是如何执行内部缓冲的，并比较两种方法是否具有相同的效果。

14.16 参考书

F. Schmidt 所著的 *The SCSI Bus and IDE Interface:Protocols, Applications, and Programming* 第 2 版。如果 SD 卡接口的简单激起了你的好奇心，那么你现在一定很想知道在个人计算机领域里使用的大部分旧的大容量存储设备（非固态）使用的接口是什么样的。你将从这本书中得知那些接口其实也没有多复杂。

Jan Axelson 所著的 *USB Mass Storage: Designing and Programming Devices and Embedded Hosts*。这本书是 Jan Axelson 关于 USB 的系列图书中的一本。如同你在本章中看到的那样，直接面向 SD/MMC 存储卡的底层接口非常简单，但是创建一个真正的针对大容量存储设备的 USB

接口则是一个复杂性成倍增长的大工程。

14.17 链接

www.mmca.org/home。多媒体卡协会（MMCA）的官方网站。

www.sdcard.org。安全数字卡协会（SDCA）的官方网站。

www.sdcard.org/Sdio/Simplified%20SDIO%20Card%20Specification.pdf。SDIO 卡规范的简化版。根据 SDIO，SD 接口不再只用于大规模存储，也可以是一些高级外围设备和小型嵌入式设备的可选接口，例如 GPS 接收器、数码相机等。

第 15 章　读写文件

15.1　计划

在上一章中，我们开发了一个基本的（软硬件）接口模块，它能用于访问 SD/MMC 卡，还能为需要大容量数据存储的应用程序提供支持。对于其他类型的大容量存储介质，也可以开发类似的接口，但是在本章中我们准备把重点放在大容量存储设备与主流 PC 操作系统（DOS、Windows 和一些 Linux 版本）共享信息所需的算法和数据结构上面。也就是说，我们将开发一个访问 FAT16 标准文件系统的接口模块。

第一个 FAT 文件系统是 1977 年由比尔·盖茨和马克·麦克唐纳创建的，用于管理 Microsoft Disk BASIC 中的磁盘。当时它使用的是此前已有文件系统中就已经使用过的技术，而在接下来的几十年里，则逐渐演变了多个版本来适应更大容量的存储设备，并开发了一些新的特征。在目前仍在使用的众多版本之中，FAT12、FAT16 和 FAT32 格式是最常见的。特别是 FAT16 和 FAT32，能够被当前的所有 PC 操作系统识别；访问效率和存储介质容量通常是在这两种格式间进行取舍的决定因素。最终，在消费类多媒体应用中常见的 Flash 格式的大容量存储设备选择了 FAT16 作为文件系统的格式。

15.2　准备

在本章中，我们将继续使用上一章中使用的硬件平台。此外还需要一个 Explorer 16 演示板或者其他同类演示板，并且必须带有一个额外的扩展板或原型电路，用于连接 SD 卡连接器以及一些上拉电阻。在本书配套网站（www.exploringPIC32.com）上可以找到有关扩展板的更多信息，了解这些信息有助于更好地完成本章中的实验。

15.3　探索

FAT 是 File Allocation Table（*文件分配表*）的首字母缩写，它也是在该文件系统中使用的最重要的数据结构之一的名称。毕竟，文件系统只是存储和组织计算机文件以及它们所包含的数据、使其便于寻找和访问的一种方法。然而，在个人计算机的发展历史中，标准和技术通常是不断进化演变得来的，而并非原始创造。基于这个原因，在接下来的讨论中将要讲述的关于 FAT 文件系统的大部分详细信息，都只能在特定的背景下进行阐述。为了与众多遗留技术与软件保持兼容，FAT 文件系统多年来一直都在为之斗争，每一种特定背景都与某一种斗争联系在一起。

15.4　扇区和簇

FAT 文件系统底层的基本原理也很简单。如同我们在前面的几章中看到的那样，大部分大容量存储设备都遵循了源自于硬盘的管理技术，即把存储空间分成固定大小的块进行管理，每个块大小为 512B，通常被称为一个*扇区*（sector）。在 FAT 文件系统中，有一小部分扇区是保留的，它们用于存储通用索引（即文件分配表）剩下的（大部分）扇区则用于存储常规数据。但是扇区并不是一一单独处理的，而是将连续扇区进行分组，形成新的更大的块，称之为*簇*（cluster）。簇可以小到只有一个扇区，也可以大到包含 64 个扇区。文件分配表是根据簇的用途

和位置来查找的。因此，簇才是 FAT 文件系统中存储器的真正最小单元（如图 15-1 所示）。

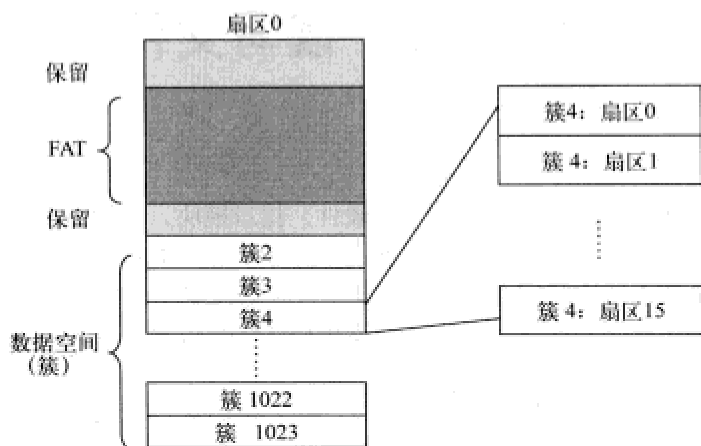


图 15-1 简化的 FAT 文件系统布局示例

图 15-1 中的简化原理图给出了一个假想的格式化为 1022 个簇的 FAT 文件系统示例，每个簇包含 16 个扇区。（注意，数据区总是从 2 号扇区开始。）在本例中，每个簇包含 8KB 的数据，总的存储空间则大约为 8MB。

簇越大，管理整个存储空间的难度越小，分配表所需的空间越小，因而文件系统的效率也越高。反过来说，如果需要写入很多小文件，那么簇越大，浪费的空间就越多。把存储设备格式化为 FAT 文件系统来使用时，通常由操作系统负责权衡，设置一个理想的簇大小值。

15.5 文件分配表

在 FAT16 文件系统中，文件分配表本质上是一个 16 位整数数组。数组中的每个元素表示一个簇。如果某个簇是空的，那么表中相应条目的取值为 0x0000。如果某个簇已被占用，并且包含某个文件的所有数据，那么对应条目的取值为 0xFFFF。如果文件体积大于单个簇的大小，那么就会形成一个簇的链表。文件分配表中的每个元素包含链表中下一个簇的索引。链表中最后一个簇的对应条目值设置为 0xFFFF。

另外，保留簇用特殊值 0x0001 来标识，而坏簇则用 0xFFF7 来标识。因为 0x0000 和 0x0001 已经赋予了特殊含义（分别表示空闲和保留），这也就解释了为什么惯例上数据区总是从 2 号簇开始进行计数。文件分配表中对应的前两个条目也同样保留。

从图 15-2 中可以看到对应于图 15-1 中示例文件系统的分配表。簇 0 和簇 1 保留。簇 2 看起来包含了一些数据，并且所有 16 个扇区或者其中某些扇区已填充了文件数据，这个文件的体积一定小于 8KB。

簇 3 看起来是链表里 3 个簇中的第一个簇，另外两个簇是簇 4 和簇 5。簇 3 和簇 4 中的所有扇区，以及簇 5 中的部分扇区已经填充了文件数据，该文件的大小超过 16KB，但小于 24KB（目前只能是猜测）。剩下的所有簇看起来都是空闲且可用的。

注意文件分配表本身的大小为簇的总数乘以 2（每个簇需要 2B），它可以占据多个扇区。在前一个示例中，一个针对 1024 个簇的文件分配表需要 2048B，或者说 4 个扇区，每个扇区为 512B。另外，因为文件分配表算是整个 FAT 文件系统中最关键的数据结构了，所以需要保存多个副本（通常为 2 个），并且会在数据空间之前按顺序依次给每个副本分配空间。

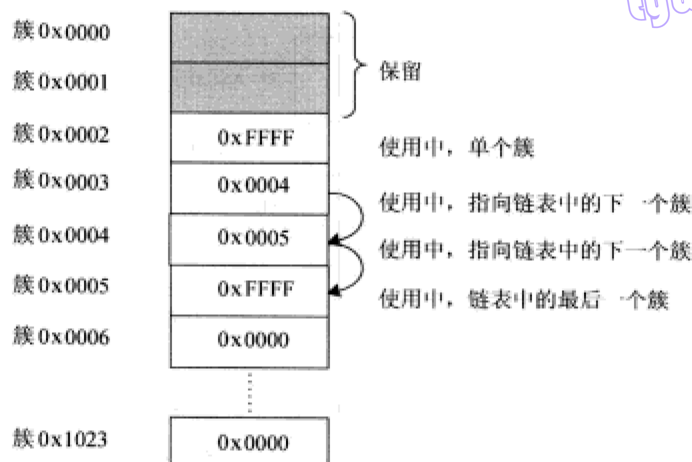


图 15-2 文件分配表中的簇链表

15.6 根目录

文件分配表的任务是跟踪数据的分配时间和分配空间情况。它不包含数据所属文件本身的任何信息。为了掌握文件信息，需要另一个叫做*根目录*（root directory）的数据结构，其用途是存储文件名、文件大小、日期、时间和其他一些属性信息。在 FAT16 文件系统中，给根目录（以下也简称为根）分配了固定大小的存储空间，位于文件分配表（第二个副本）和第一个数据簇（2 号簇）之间的固定位置，如图 15-3 所示。

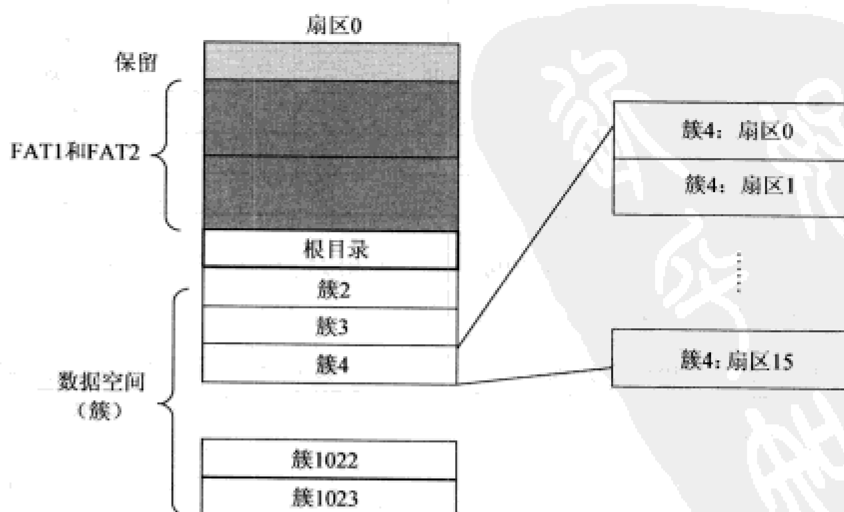


图 15-3 FAT 文件系统布局示例

因为位置和大小（扇区数目）都是固定的，所以根目录中的最大文件数目（或者说是目录项）是受限的，并且在存储设备格式化时就已确定。分配给根的每个扇区允许包含 16 个文件条目，每个条目需要 32B 的存储空间，如图 15-4 所示。

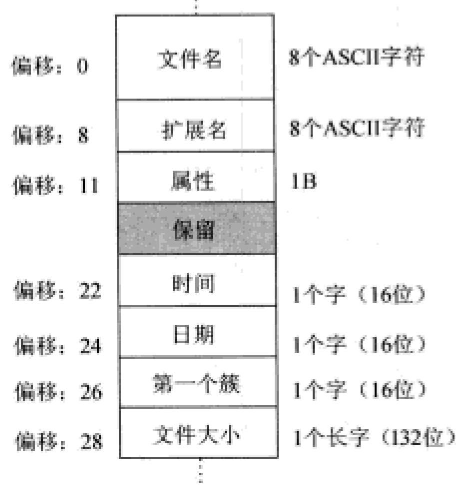


图 15-4　根目录条目基本结构

如果你对旧的使用 8：3 命名方式的微软操作系统很熟悉，那么文件名和扩展名字段是最容易理解的。这两个字段只需把完整的文件名中的点去掉并用空格分开即可。

属性字段由一组标志位构成，这些标志位的含义如表 15-1 所示。

表 15-1　目录条目中的文件属性

位	掩　码	描　述
0	0x01	只读
1	0x02	隐藏
2	0x04	系统
3	0x08	卷标
4	0x10	子目录
5	0x20	归档

时间和日期字段是指文件最后被修改的时间，必须以特殊格式进行编码，从而将所有信息压缩到 2 个 16 位字中去（参见表 15-2 以及表 15-3）。

表 15-2　目录条目字段中的时间编码

位	描　述
15~11	小时（0~23）
10~5	分钟（0~59）
4~0	秒/2（0~29）

表 15-3　目录条目字段中的日期编码

位	描　述
15~9	年（0=1980，127=2107）
8~5	月（1=1 月，12=12 月）
4~0	日（1~31）

注意在日期字段编码中，0x0000 是不能表示合法日期的。当该字段没有使用或者可能已经损坏时，这可以给文件系统提供一些线索。

第一个簇字段提供了和 FAT 表的基本联系。这个 16 位字中的内容是包含文件数据的第一个簇（可能也是唯一的一个簇）的编号。

最后，文件大小字段是一个 32 位的整数，给出了文件数据的大小（以字节表示）。

根据目录条目中文件名的第一个字符，我们也可以判断该条目当前是否正在使用以及是如何使用的。

- 如果它包含一个可打印的 ASCII 字符，那么该条目有效，并正在使用中。
- 如果它是 0，那么表示该条目为空。如果浏览整个目录，也可以推断出文件列表是终止于此的，因为文件系统是严格按顺序使用目录表中的所有条目的。

当文件从目录中移除时还有第三种可能性。在这种情况下，会直接用一个特殊码（`0xE5`）替换文件名的第一个字符。这表示条目内容不再合法，新的文件可以继续使用该条目。那么，在通过搜索根目录来查找文件时，不能在遇到 `0xE5` 时就终止搜索，因为后面可能还有更多的有效条目。

如果要对 FAT16 文件系统的结构进行完整说明，那还需要很多讲解。但是如果你已经掌握了目前介绍的这些内容，那么可以说对其核心机制已经有了适当了解。你可以准备好全力以赴地学习下面的知识了，因为我们马上就开始编写代码了。

15.7 寻宝

到目前为止，我们对文件系统的描述一直带有一定程度的简化，因为我们忽略了一些基本问题，如下所示。

- 从何处获知存储设备的总容量？
- 如何得知 FAT 的位置？
- 如何得知每个簇包含多少个扇区（1~64）？
- 如何得知数据空间的开始位置？

上述所有问题的答案很快就会揭晓，但是揭晓过程更像一个寻宝的过程，而不是依靠逻辑分析一步一步得出的。实际上，在图 15-5 中就能发现第一组线索。通过解释这些线索，我们将逐步创建出一个新函数，该函数能挂载（mount）文件系统并解密其内容，这就是我们要寻找的宝藏。

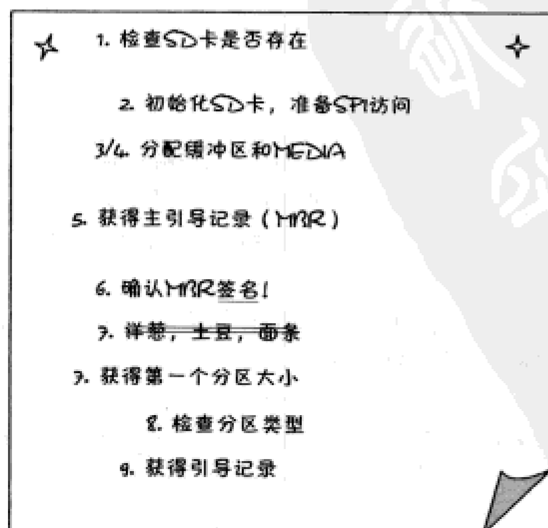


图 15-5　第一组线索

使用第 14 章中开发的 SDMMC.c 模块函数，用 `initSD()` 函数初始化 I/O，检查槽中存储

卡是否存在。

```
// 0. init the I/Os
initSD();

// 1. check if the card is in the slot
if (!detectSD())
{
  FError = FE_NOT_PRESENT;
  return NULL;
}
```

继续用 initMedia() 函数初始化 SD 卡，使其工作在 SPI 模式下。

```
// 2. initialize the card
if ( initMedia())
{
  FError = FE_CANNOT_INIT;
  return NULL;
}
```

同时使用标准 C 函数库中的 malloc() 函数动态分配两个数据结构：

```
// 3. allocate space for a MEDIA structure
D = (MEDIA *) malloc( sizeof( MEDIA));
if ( D == NULL)              // report an error
{
  FError = FE_MALLOC_FAILED;
  return NULL;
}

// 4. allocate space for a temp sector buffer
buffer = (unsigned char *) malloc( 512);
if ( buffer == NULL)       // report an error
{
  FError = FE_MALLOC_FAILED;
  free( D);
  return NULL;
}
```

第一个数据结构是 MEDIA。在之后的内容中会详细解释该函数，但现在只需要知道它就是存放我们众多答案的宝库所在就足够了。也许 CHEST 这个名称更合适？

第二个数据结构是 buffer，它就是一个简单的 512B 的大数组，用于在寻宝过程中检索数据扇区。

注意，为了使 malloc() 函数能成功地分配存储空间，用户必须记住设置 MPLAB C32 链接器，使其保留一部分 RAM 空间作为堆使用。



提示　查看 Build Project（ICT）检查表可以学习如何修改链接器的设置。

所有大容量存储设备的第一个扇区（LBA 0）都用来存储主引导记录（master boot record, MBR），这是由很多历史原因决定的。

以下代码说明了如何第一次调用 readSECTOR() 函数来访问 MBR。

```
// 5. get the Master Boot Record
if ( !readSECTOR( 0, buffer))
{
  FError = FE_CANNOT_READ_MBR;
  free( D); free( buffer);
  return NULL;
}
```

MBR 扇区中最后一个字中包含的特征码（0x55AA），表明我们确实读到了正确的数据。

```
#define FO_SIGN     0x1FE  // MBR signature location (55,AA)

  // 6. check if the MBR sector is valid
  //    verify the signature word
  if (( buffer[ FO_SIGN] != 0x55) ||
      ( buffer[ FO_SIGN +1] != 0xAA))
  {
    FError = FE_INVALID_MBR;
    free( D); free( buffer);
    return NULL;
  }
```

以前，MBR 中通常包含 PC 在上电时会实际执行的一段代码。但现在已经没有计算机这样做了，当然对于我们的 PIC32 应用来说，这段 8086 代码也是没有用的。大多数时候你会发现 MBR 几乎全部用 0 来填充，只是一些曾经用于存储关键信息的位置不为 0（如图 15-6 所示）。

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	Access
00000000	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000010	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000030	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000040	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000050	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000060	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000070	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000080	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000090	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000000A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000000B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000000C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000000D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000000E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000000F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000100	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000110	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000120	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000130	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000140	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000150	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000160	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000170	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000180	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
00000190	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000001A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000001B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	03	....
000001C0	35	00	06	08	D8	C1	F1	00	00	00	0F	C9	0E	00	00	00	5...ØÁñ....É....
000001D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000001E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000001F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	55	AA	..............Uª

图 15-6 MBR 的内容（十六进制表示）

例如，从偏移位置 0x01BE 开始，你会发现分区表（partition table）。这个表仅由 4 个条目所组成，每个条目 16B。分区表的功能是允许单个存储设备能够驻留在多个操作系统中，并将存储空间分成多个安全区域，每个区域都作为完全独立的存储设备使用。

针对我们的应用，假定（要求）整个 SD/MMC 卡格式化为一个分区。因此，我们仅需要注意分区表中的第一个条目（16B 的存储块）。在这 16B 中，我们也只需访问其中的某些字节来获取：

- 分区大小（应该包含整个存储卡）；
- 开始扇区；
- 所包含文件系统的类型，这是最重要的。

以下两个宏从分区表读取数据，并将其转换为 16 位和 32 位字：

```
#define ReadW( a, f) *(unsigned short*)(a+f)
#define ReadL( a, f) *(unsigned short*)(a+f)+\
                     (( *(unsigned short*)(a+f+2))<<16)
```

另外，以下定义会给出正确的 MBR 偏移地址。

```
//-------------------------------------------------------------
// Master Boot Record key fields offsets
#define FO_MBR          0L     // master boot record sector LBA
#define FO_FIRST_P      0x1BE  // offset of first partition table
#define FO_FIRST_TYPE   0x1C2  // offset of first partition type
#define FO_FIRST_SECT   0x1C6  // first sector of first partition
#define FO_FIRST_SIZE   0x1CA  // number of sectors in partition
#define FO_SIGN         0x1FE  // MBR signature location (55,AA)

  // 7. read the number of sectors in partition
  psize = ReadL( buffer, FO_FIRST_SIZE);

  // 8. check if the partition type is acceptable
  i = buffer[ FO_FIRST_TYPE];
  switch ( i)
  {
    case 0x04:
    case 0x06:
    case 0x0E:
      // valid FAT16 options
      break;
    default:
      FError = FE_PARTITION_TYPE;
      free( D); free( buffer);
      return NULL;
  } // switch
```

基于一些历史原因，不同的分区类型是用不同的特殊码来表示的。我们必须能正确译码至少 3 种类型的 FAT16 分区，包括 0x04、0x06 和 0x0E。

访问 MBR 并寻找分区表，有点儿像拿到一张需要解释的地图，而这张地图采用全新的符号来表示，其中给出的线索也是陌生的（如图 15-7 所示）。

把从偏移地址 FO_FIRST_SECT（0x1C6）处找到的 32 位字提取出来，它是第一个分区（根据我们的假设，也是唯一的分区）的分区表条目的一部分，从中我们可以获得该分区中第一个扇区的地址（LBA）。

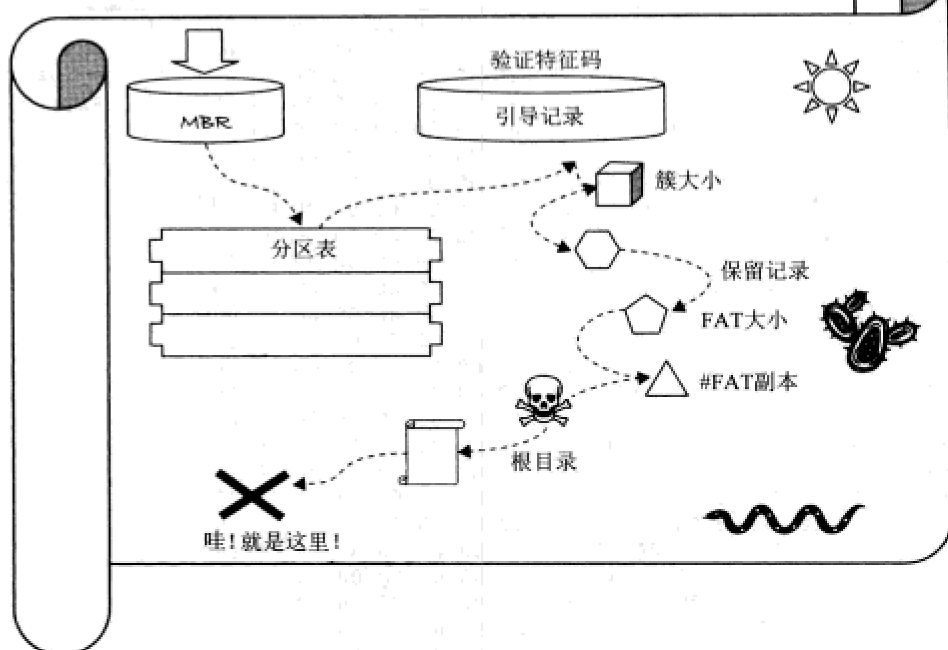


图 15-7　地图

```
// 9. get the first partition first sector -> Boot Record
firsts = ReadL( buffer, FO_FIRST_SECT);

// 10. get the sector loaded (boot record)
if ( !readSECTOR( firsts, buffer))
{
   free( D); free( buffer);
   return NULL;
}
```

它也有一个和 MBR 类似的特征码，位于扇区的最后一个字中，在继续后续工作之前需要对其进行验证。

```
// 11. check if the boot record is valid
//      verify the signature word
if (( buffer[ FO_SIGN] != 0x55) ||
   ( buffer[ FO_SIGN +1] != 0xAA))
{
   FError = FE_INVALID_BR;
   free( D); free( buffer);
   return NULL;
}
```

这个扇区被称为（第一个分区的）引导记录（boot record），并且它也包含现在已经毫无用处的可执行代码（如图 15-8 所示）。

幸运的是，在这个地址固定的记录里，同样也包含了更多我们正在寻找的答案，以及可以帮助我们搜寻整个 FAT16 文件系统地图的新线索。以下给出的都是包含在引导记录缓冲区中

的关键偏移值：

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	Access
0001E200	EB	00	90	20	20	20	20	20	20	20	20	00	02	20	01	00	ë.▮ .. ..
0001E210	02	00	02	00	00	F8	77	00	3F	00	10	00	F1	00	00	00	.....øw.?...ñ...
0001E220	0F	C9	0E	00	80	00	29	13	18	FD	E0	20	20	20	20	20	.É..▮.)..ýà
0001E230	20	20	20	20	20	20	46	41	54	31	36	20	20	20	00	00	FAT16 ..
0001E240	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0001E250	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0001E260	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0001E270	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0001E280	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0001E290	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0001E2A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0001E2B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0001E2C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0001E2D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0001E2E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0001E2F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0001E300	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0001E310	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0001E320	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0001E330	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0001E340	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0001E350	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0001E360	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0001E370	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0001E380	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0001E390	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0001E3A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0001E3B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0001E3C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0001E3D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0001E3E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
0001E3F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	55	AA	..............Uª

图 15-8 引导记录的内容（十六进制表示）

```
// Partition Boot Record key fields offsets
#define BR_SXC        0xd     // (byte) sectors per cluster
#define BR_RES        0xe     // (word) reserved sectors
#define BR_FAT_SIZE   0x16    // (word) FAT size in sectors
#define BR_FAT_CPY    0x10    // (byte) number of FAT copies
#define BR_MAX_ROOT   0x11    // (odd word) max entries in root
```

根据以下代码，可以计算一个簇的大小：

```
// 12. determine the size of a cluster
D->sxc = buffer[ BR_SXC];
// this will also act as flag that the media is mounted
```

确定 FAT 的位置、大小以及副本数量：

```
// 13. determine fat, root and data LBAs
// FAT = first sector in partition (boot record)
//      +reserved records
D->fat = firsts + ReadW( buffer, BR_RES);
D->fatsize = ReadW( buffer, BR_FAT_SIZE);
D->fatcopy = buffer[ BR_FAT_CPY];
```

也能找到根目录的位置：

```
// 14. ROOT = FAT + (sectors per FAT * copies of FAT)
D->root = D->fat + ( D->fatsize * D->fatcopy);
```

但现在需要注意了！在我们就要完成最后几步的时候，小心陷阱！

```
// 15. MAX ROOT is the maximum number of entries
//in the root directory
D->maxroot = ReadW( buffer, BR_MAX_ROOT) ;
```

你明白了吗？没有？那好，给你一个提示：看看在前面几行中定义的 BR_MAX_ROOT 偏移量的值。你会发现这是一个奇数地址（0x11）。这就是 ReadW() 宏所做的事情，它试图把这个奇数地址作为字地址使用，这必然导致 PIC32 抛出一个处理器异常（未对齐的字访问），从而进入异常处理！

因此，我们需要一个特殊的宏（也许并不高效），一次一个字节地组合成一个字，从而保证不会掉入陷阱！

```
// this is the safe versions of ReadW to be used on odd
address fields
#define ReadOddW( a, f) (*(a+f) + ( *(a+f+1) << 8))

  // 15. MAX ROOT is the maximum number of entries
  //      in the root directory
  D->maxroot = ReadOddW( buffer, BR_MAX_ROOT) ;
```

剩下的两条信息是很容易被捕获到的。利用它们可以得知数据区域（按照簇进行划分）是从哪里开始的，以及有多少个簇可用：

```
// 16. DATA = ROOT + (MAXIMUM ROOT *32/512)
D->data = D->root + ( D->maxroot >> 4);
// assuming maxroot % 16 == 0!!!

// 17. max clusters in this partition
//      = (tot sectors - sys sectors )/sxc
D->maxcls = (psize - (D->data-firsts))/D->sxc;
```

我们花了多达 17 个步骤来获得宝藏，现在已经拥有了画出整个 FAT16 文件系统布局的所有信息。SD/MMC 存储卡采用的是 FAT16 文件系统，其实任何其他根据 FAT16 标准进行格式化的大容量存储设备上采用的都是 FAT16 文件系统。我们获得的宝藏，说穿了，也无非就是另外一张地图，一张从现在开始用于在大容量存储设备上寻找文件的地图（如图 15-9 所示）。

现在需要把我们辛辛苦苦找来的所有信息组织到一起。我们使用一开始就已分配到堆上的 MEDIA 数据结构。

```
typedef struct {
  LBA    fat;             // lba of FAT
  LBA    root;            // lba of root directory
  LBA    data;            // lba of the data area
  unsigned maxroot;       // max entries in root
  unsigned maxcls;        // max clusters in partition
  unsigned fatsize;       // number of sectors
  unsigned char fatcopy;  // number of FAT copies
  unsigned char sxc;      // number of sectors per cluster
  } MEDIA;
```

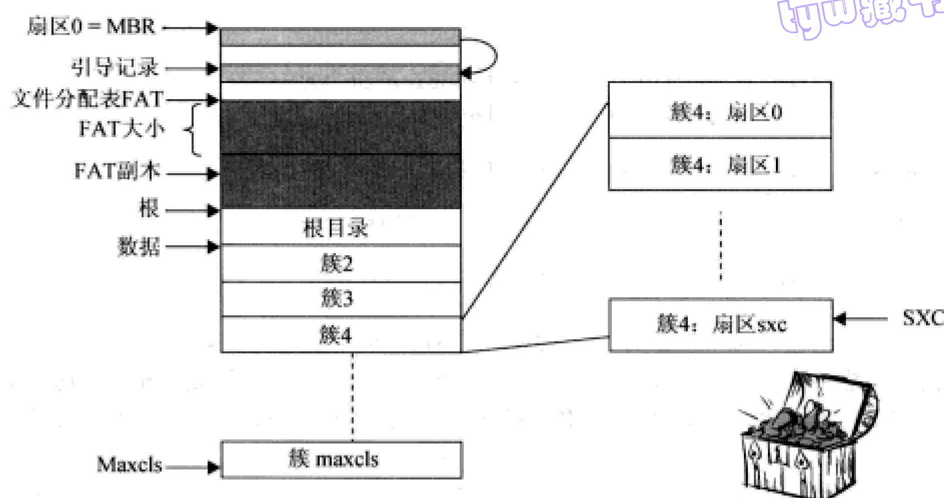


图 15-9　FAT16 的完整布局

把刚才开发的所有代码集中到一起，形成一个 mount() 函数。这个名称听起来和面向 Linux 系列的操作系统编程中经常遇到的术语非常类似。

对于 Linux 系统下的大容量存储设备来说，它必须首先挂载（mount）到文件系统中，换句话说，作为新的分支附加到主（系统）文件系统中去。Windows 用户可能并不熟悉这个概念，因为他们并不需要选择是否、何时、何地挂载新的设备文件系统。所有新的大容量存储设备都在上电时自动并且无条件地挂载到 Windows 中；那些可移除的存储介质插入到插槽中以后，则会在 Windows 文件系统的最根部赋给它一个唯一的、单字母的标识符（C:、D:、E: 等）。

```
MEDIA * mount( void)
{
  LBA psize;        // number of sectors in partition
  LBA firsts;       // first sector inside the first partition
  int i;
  unsigned char *buffer;

  ... insert here all 17 steps of our treasure hunt

  // 18. free up the temporary buffer
  free( buffer);
  return D;

} // mount
```

再定义一个全局指针 D，指向 MEDIA 结构。它是整个文件系统的根本，对目前来说，在任何时候都只有一个存储设备可用（一个连接器/插槽、一个卡）。

```
// global definitions
MEDIA *D;
```

我们也需要定义一个 unmount() 函数，用于释放 MEDIA 数据结构所占用的存储空间。

```
void unmount( void)
{
  free( D);
} // unmount
```

15.8 打开文件

现在我们已经揭开了 FAT16 文件系统的所有秘密，于是就可以回到我们最初的目标了：访问单个文件并和 PC 共享。本节将开发一组和大多数操作系统中文件操作函数类似的高级函数。我们需要一个用于在存储设备上寻找文件位置的函数，一个用于从文件中顺序读取数据的函数，可能还需要一个写数据和创建新文件的函数。

按照逻辑顺序，首先开发名为 fopenM()的函数。该函数用于寻找某个文件（如果存在）的所有信息，并将其集中到一个新的 MFILE 数据结构里。



注解 这个数据结构名称的选择是为了避免和标准 C 库文件 stdio.h 中定义的类似结构和函数冲突。

```
typedef struct {
  MEDIA * mda;                  // media structure pointer
  unsigned char * buffer;       // sector buffer
  unsigned short cluster;       // first cluster
  unsigned short ccls;          // current cluster in file
  unsigned short sec;           // sector in current cluster
unsigned short pos;             // position in current sector
unsigned short top;             // bytes in the buffer
int    seek;                    // position in the file
int    size;                    // file size
unsigned short time;            // last update time
unsigned short date;            // last update date
char    name[11];               // file name
char    mode;                   // mode 'r', 'w'
unsigned short fpage;           // FAT page currently loaded
unsigned short entry;           // entry position in cur dir
} MFILE;
```

我知道，乍一看这个数据结构很庞大，超过了 40B，但是随着讨论的深入，你将发现所有这些定义都是必需的。从现在开始你必须得相信我。

模仿标准 C 库的实现（对很多操作系统都是通用的），fopenM()函数将包含 2 个（ASCII）字符串参数：文件名和一个值为 r 或 w 的“模式”字符串，用于指示文件打开以后是用于读取还是写入。

```
MFILE *fopenM( const char *filename, const char *mode)
{
  char c;
  int i, r, e;
  unsigned char *b;
  MFILE *fp;
```

为了优化存储器使用，我们仅在需要时才为 MFILE 结构分配空间，而这也正是 fopenM()函数的第一个任务。其返回值即为指向该数据结构的指针。如果 fopenM()函数执行失败，则返回 NULL 指针。

当然，得到存储设备文件系统的布局是打开某个文件的先决条件，而这是 mount()函数的任务。一个指向 MEDIA 结构的指针必须已经包含在全局指针 D 中了。

```
  // 1. check if a storage device is mounted
  if ( D == NULL)    // unmounted
```

```
    {
    FError = FE_MEDIA_NOT_MNTD;
    return NULL;
  }
```

因为存储设备的所有行为都必须以 512B 的存储块为单位来进行，所以我们需要分配这么大的一个空间，作为读/写缓冲区使用。

```
// 2. allocate a buffer for the file
b = (unsigned char*)malloc( 512);
if ( b == NULL)
{
  FError = FE_MALLOC_FAILED;
  return NULL;
}
```

只有在可以提供这个存储空间的前提下，才能继续为 MFILE 结构分配其他存储空间。

```
// 3. allocate a MFILE structure on the heap
fp = (MFILE *) malloc( sizeof( MFILE));
if ( fp == NULL)        // report an error
{
  FError = FE_MALLOC_FAILED;
  free( b);
  return NULL;
}
```

buffer 指针和 MEDIA 指针现在都可以记录在 MFILE 结构里面。

```
// 4. set pointers to the MEDIA structure and buffer
fp->mda = D;
fp->buffer = b;
```

必须解析文件名参数，把每个字符转换为大写（使用定义在 ctype.h 中的标准 C 库函数），如果文件名长度不满 8 个字符，就必须用空格把剩余的位都填满。

```
// 5. format the filename into name
for( i=0; i<8; i++)
{
  // read a char and convert to upper case
  c = toupper( *filename++);
  // extension or short name noextension
  if (( c == '.') || ( c == '\0'))
    break;
  else
    fp->name[i] = c;
} // for
// if short fill the rest up to 8 with spaces
while ( i<8) fp->name[i++] =' ';
```

扩展名的长度最多为 3 个字符，同样也必须把它格式化并用空格进行填充。

```
// 6. if there is an extension
if ( c != '\0')
{
  for( i=8; i<11; i++)
  {
    // read char, convert to upper case
    c = toupper( *filename++);
```

```
      if ( c == '.')
        c = toupper( *filename++);
      if ( c == '\0')         // short extension
        break;
      else
        fp->name[i] = c;
    } // for
    // if short fill the rest up to 3 with spaces
    while ( i<11) fp->name[i++] = ' ';
  } // if
```

尽管大部分C库函数都提供多种文件访问模式，比如区分文本文件和二进制文件，并提供“追加”选项，但至少是在现阶段，我们只接受两个基本的模式选项：r 和 w。

```
// 7. copy the file mode character (r, w)
if ((*mode == 'r')||(*mode == 'w'))
  fp->mode=*mode;
else
{
  FError = FE_INVALID_MODE;
  goto ExitOpen;
}
```

在正确地格式化文件名以后，需要搜索存储设备的根目录，获取具有相同名称的条目信息。

```
// 8. Search for the file in current directory
if ( ( r=findDIR( fp)) == FAIL)
{
  FError=FE_FIND_ERROR;
  goto ExitOpen;
}
```

现在让我们暂时不要理会搜索细节，而相信 findDIR() 函数可以返回3个可能值：FAIL、NOT_FOUND 和 FOUND。我们必须考虑失败的可能性。毕竟，在我们考虑处理存储设备出现的致命错误之前，总是会有用户在没有确认的情况下就从插槽中把卡拔走的事情发生。如果这种事情发生，那么就和之前遇到的错误情况一样，后续工作无法进行。这时最好是立即释放目前已分配的存储空间，并且把错误码放入专门的“信箱”FError 中，然后返回一个 NULL 指针，就如同我们在挂载过程中所做的工作一样。

如果搜索文件的过程成功完成（不管是否找到），那就可以继续初始化 MFILE 结构。

```
// 9. init all counters to the beginning of the file
fp->seek = 0;           // first byte in file
fp->sec = 0;            // first sector in the cluster
fp->pos = 0;            // first byte in sector/cluster
```

在按顺序访问文件内容时，计数器 seek 用于跟踪当前位置。它是一个32位的整数（无符号），其值位于0和文件总大小（以字节计算）之间。

sec 字段用于跟踪当前簇中正在操作的扇区。它的值是一个介于 0 和 sxc-1 之间的整数，表示数据簇中的扇区编号。pos 字段则用于跟踪当前缓冲区中下一个要访问的字节。它的值是一个介于0~511的整数。

```
// 10. depending on the mode (read or write)
if ( fp->mode == 'r')
{
```

程序进行到此处，有可能是需要打开一个现有文件并读取，也有可能是需要创建一个新文

件并写入，那就需要根据不同的情况分别进行处理。现在我们先完成在读取模式（r）下调用 fopenM()函数必需的所有步骤，这时候最好是能找到这个文件。

```
// 10.1 'r' open for reading
if ( r == NOT_FOUND)
{
  FError = FE_FILE_NOT_FOUND;
  goto ExitOpen;
}
```

如果确实找到了这个文件，那么我们确信 findDIR()函数就已经向 MFILE 结构的更多字段中填入了数据，包括：

- Entry，表示根目录中该文件信息所在位置；
- Cluster，表示从目录条目中得到的存储文件数据的第一个数据簇的编号；
- Size，表示整个文件包含的字节数；
- Time 和 Date，文件创建的日期和时间；
- 文件属性。

第一个簇的编号将成为当前簇：ccls。

```
else
{ // found
// 10.2 set current cluster pointer on first cluster
  fp->ccls=fp->cluster;
```

现在我们已经获取了识别第一个数据扇区所需的所有信息。函数 readDATA()将执行简单的计算，把 ccls 和 sec 值转换成数据区内的绝对扇区号，并使用底层 readSECTOR()函数从存储设备中检索数据，稍后将详细介绍 read DATA ()。

```
// 10.3 read a sector of data from the file
  if ( !readDATA( fp))
  {
      goto ExitOpen;
  }
```

因为文件长度并没有限制为扇区大小的倍数，所以很有可能检索到缓冲区中的数据并不全部属于当前文件。MFILE 结构中的字段 top 用于跟踪当前文件的数据结束位置并进行必要的填充工作。

```
  // 10.4 determine how much data is really inside buffer
    if ( fp->size-fp->seek<512)
        fp->top=fp->size-fp->seek;
    else
        fp->top=512;
  } // found
} // 'r'
```

以上就是 fopenM()函数所需完成的所有工作，这样在打开一个文件并进行读取时，就可以返回指向 MFILE 结构的指针了。

```
// 12. Exit with success
return fp;
```

如果上面给出的任何一个步骤出现错误，都会释放分配给扇区缓冲区和 MFILE 结构的存储器空间，返回 NULL 指针并退出函数。

```
  // 12. Exit with error
ExitOpen:
  free( fp->buffer);
  free( fp);
  return NULL;
} // fopenM
```

按照从上到下的方式，现在我们来完成开发 fopenM() 过程中所需的两个辅助函数，第一个是 readDATA()：

```
unsigned readDATA( MFILE *fp)
{
  LBA l;

  // calculate lba of cluster/sector
  l = fp->mda->data+(LBA)(fp->ccls-2) * fp->mda->sxc+fp->sec;
  fp->fpage = -1;          // invalidate FAT cache

  return( readSECTOR( l, fp->buffer));
} // readDATA
```

暂时忽略 fpage 字段，注意我们是如何使用 MEDIA 数据结构中的 data 和 sxc 字段来计算目标数据扇区的绝对地址（LBA）的。非常简单！

用同样的方法创建 readDIR() 函数，从根目录读取包含给定条目的一个扇区数据。

```
unsigned readDIR( MFILE *fp, unsigned e)
// loads current entry sector in file buffer
// returns    FAIL/TRUE
{
  LBA l;

  // load the root sector containing the DIR entry "e"
  l = fp->mda->root + (e >> 4);
  fp->fpage = -1;            // invalidate FAT cache

  return ( readSECTOR( l, fp->buffer));
} // readDIR
```

每个目录条目大小为 32B，因此每个扇区包含 16 个条目。

findDIR() 函数的开发过程很快，只需要一个包含几个步骤的循环，在根目录中的所有可用条目中进行搜索即可。

```
unsigned findDIR( MFILE *fp)
// fp       file structure
// return  found/not_found/fail
{
  unsigned eCount;            // current entry counter
  unsigned e;                 // current entry offset
  int i, a;
  MEDIA *mda = fp->mda;

  // 1. start from the first entry
  eCount = 0;

  // load the first sector of root
  if ( !readDIR( fp, eCount))
      return FAIL;
```

首先加载包含前 16 个条目的第一个根扇区到缓冲区中。对于每个条目，计算其在缓冲区

内的偏移。

```
// 2. loop until you reach the end or find the file
while ( 1)
{
  // 2.0 determine the offset in current buffer
    e = (eCount&0xf) * DIR_ESIZE;
```

检查条目中文件名的第一个字符。

```
// 2.1 read the first char of the file name
  a = fp->buffer[ e + DIR_NAME];
```

如果其值为 0，表示当前条目为空并到达列表尾部，可以立即报告文件名没找到，并退出程序。

```
// 2.2 terminate if it is empty (end of the list)
  if ( a == DIR_EMPTY)
  {
    return NOT_FOUND;
  } // empty entry
```

另一个可能性是该条目被标记为已删除，这种情况下应该跳过该条目并继续搜索。

```
// 2.3 skip erased entries if looking for a match
  if ( a != DIR_DEL)
  {
```

否则，它就是一个有效的正确条目，我们应该检查其属性，从而确定它是否对应于正确的文件或者对应于其他类型的目标，如：

- 子目录；
- 卷标；
- 长文件名。

这些都不是我们所关心的，因为我们不想把问题复杂化，所以会尽量避开最新的 FAT 文件系统标准所提供的一些先进而又有专利保护的特性。

```
// 2.3.1 if not VOLume or DIR compare the names
  a = fp->buffer[ e + DIR_ATTRIB];
  if ( !(a & (ATT_DIR | ATT_HIDE)) )
  {
```

接下来一个字符接一个字符地比较文件名，寻找完全匹配。

```
// compare file name and extension
for (i=DIR_NAME; i<DIR_ATTRIB; i++)
{
  if ( fp->buffer[ e+i] != fp-> name[i])
    break; // difference found
}
```

只有当每个字符都匹配时，我们才从条目中提取出有用的信息，复制到 MFILE 结构中去，并返回 FOUND 值。

```
      if ( i == DIR_ATTRIB)
      {
        // entry found, fill the file structure
        fp->entry = eCount;      // store index
        fp->time = ReadW( fp->buffer, e+DIR_TIME);
        fp->date = ReadW( fp->buffer, e+DIR_DATE);
```

```
            fp->size = ReadL( fp->buffer, e+DIR_SIZE);
            fp->cluster = ReadL( fp->buffer, e+DIR_CLST);
            return FOUND;
        }
    } // not a dir nor a vol
} // not deleted
```

如果文件名和扩展名不匹配，那么就继续搜索下一个条目，记住每过 16 个条目就需要从根目录中加载下一个扇区。

```
// 2.4 get the next entry
  eCount++;
  if ( (eCount & 0xf) == 0)
  {  // load a new sector from the Dir
      if ( !readDIR( fp, eCount))
        return FAIL;
  }
```

根目录中的最大条目数（maxroot）是已知的，如果到达了目录尾部也没有找到匹配的文件名，就终止搜索并返回 NOT_FOUND。

```
      // 2.5. exit the loop if reached the end or error
        if ( eCount >= mda->maxroot)
          return NOT_FOUND;     // last entry reached
    }// while
} // findDIR
```

15.9 从文件中读取数据

终于到了我们等待已久的时刻了。文件系统已挂载，文件已找到并打开，等待读取。现在需要开发一个 freadM() 函数，从文件中自由地读取数据块。

```
unsigned freadM( void * dest, unsigned size, MFILE *fp)
// fp        pointer to MFILE structure
// dest      pointer to destination buffer
// count     number of bytes to transfer
// returns   number of bytes actually transferred
{
  MEDIA * mda = fp->mda;
  unsigned count=size;            // counts bytes to be transfer
  unsigned len;
```

文件名、读取数量以及参数传递的顺序还是模仿标准 C 库中类似名称的函数来设置。另外还提供一个目标缓冲区参数，把从文件中读取的数据复制进去，当常规指针被传递给已开放的 MFILE 结构时，字节传输请求也同时开始。

freadM() 函数会从文件中读取尽可能多的字节，并返回一个无符号整数，说明读取的有效字节数是多少。在我们这个简单的实现中，如果返回的数目和调用程序请求的数目不一致，那么就只能认为是发生了某些错误。很有可能已经到达了文件末尾，但是也可能是其他类型的错误发生，而我们并没有进行区分；例如，在读取过程中存储卡被拔走了。

和往常一样，我们并不信任从参数列表中传递进来的指针，因此会检查它是否真地指向合法的、初始化过的 MFILE 结构。

```
// 1. check if fp points to a valid open file structure
if (( fp->mode != 'r'))
{  // invalid file or not open in read mode
  FError = FE_INVALID_FILE;
  return 0;
}
```

如果检查合格，才能进入从扇区数据缓冲区传输数据的循环中。

```
// 2. loop to transfer the data
while ( count>0)
{
```

在循环内部，第一个要检查的条件就是当前所在位置和文件总体积之间的关系。

```
// 2.1 check if EOF reached
if ( fp->seek >= fp->size)
{
  FError=FE_EOF;    // reached the end
  break;
}
```

这个错误只有在调用 freadM()函数的应用程序忽略了以下情况时才会发生：上一次 freadM()函数调用返回的数据字节数目小于请求数目；或者，调用程序已经在前面的调用中请求过文件读取，而读取的字节数目恰好等于文件中包含的字节数目。

如果没有出错，就验证当前缓冲区中的数据是不是已经全部使用过了。

```
// 2.2 load a new sector if necessary
if (fp->pos == fp->top)
{
```

如果是，就必须重置缓冲区指针并从文件中加载下一个扇区。

```
fp->pos = 0;
fp->sec++;
```

如果已经用完了当前簇中的所有扇区，那么就必须通过查看文件分配表并沿着簇链表而进入下一个簇。

```
// 2.2.1 get a new cluster if necessary
if ( fp->sec == mda->sxc)
{
  fp->sec = 0;
  if ( !nextFAT( fp, 1))
  {
    break;
  }
}
```

不管是哪种情况，在新的扇区数据加载到缓冲区中时，都需要验证它是否是文件所在的最后一个扇区，以及是否只有部分内容属于此文件。

```
// 2.2.2 load a sector of data
if ( !readDATA( fp))
{
  break;
}
// 2.2.3 determine how much data is inside buffer
```

```
if ( fp->size-fp->seek < 512)
  fp->top = fp->size - fp->seek;
else
  fp->top = 512;
} // load new sector
```

现在已经确知数据存在于缓冲区并已准备开始传输，因此可以决定单次传输的数量。

```
// 2.3 copy as many bytes as possible in a single chunk
// take as much as fits in the current sector
if ( fp->pos+count < fp->top)
  // fits all in current sector
  len = count;
else
  // take a first chunk, there is more
  len = fp->top - fp->pos;

memcpy( dest, fp->buffer + fp->pos, len);
```

使用标准 C 函数库（string.h）中的 memcpy() 函数，将一个数据块从文件缓冲区移动到目的缓冲区，这些例程已经进行了执行速度的优化，因此我们将获得最好的性能。更新指针和计数器并重复循环，直到请求的所有数据传输完毕。

```
  // 2.4 update all counters and pointers
    count-= len;          // compute what is left
    dest += len;          // advance destination pointer
    fp->pos += len;       // advance pointer in sector
    fp->seek += len;      // advance the seek pointer

} // while count
```

最后，返回循环中实际传输的字节数量，并退出函数。

```
  // 3. return number of bytes actually transferred
  return size-count;
} // freadM
```

nextFAT() 函数用于遵循簇链表的顺序，从当前簇移动到下一个簇。

```
unsigned nextFAT( MFILE * fp, unsigned n)
// fp  file structure
// n   number of links in FAT cluster chain to jump through
//     n==1, next cluster in the chain
{
  unsigned c;
  MEDIA * mda=fp->mda;

  // loop n times
  do {
    // get the next cluster link from FAT
    c = readFAT( fp, fp->ccls);
    // compare against max value of a cluster in FATxx
    // return if eof
    if ( c >= FAT_MCLST)     // check against eof
    {
      FError=FE_FAT_EOF;
      return FAIL;  // seeking beyond EOF
    }
```

```
    // check if cluster value is valid
    if ( c >= mda->maxcls)
    {
      FError = FE_INVALID_CLUSTER;
      return FAIL;
    }

  } while (--n>0);// loop end

  // update the MFILE structure
  fp->ccls=c;

  return TRUE;
} // get next cluster
```

可见，nextFAT()函数循环使用 readFAT()函数来执行实际的加载 FAT 段（扇区）的繁重工作。

```
unsigned readFAT( MFILE *fp, unsigned ccls)
// mda       disk structure
// ccls      current cluster
// return    next cluster value,
//           0xffff if failed or last
{
  unsigned p, c;
  LBA l;

  // get page of current cluster in fat
  p = ccls >>8;                 // 256 clusters per sector

  // check if already cached
  if (fp->fpage != p)
  {
    // load the fat sector containing the cluster
    l = fp->mda->fat + p;

    if ( !readSECTOR( l, fp->buffer))
      return FAT_EOF;          // failed
   // note the sector contains a valid FAT page cache
   fp->fpage = ccls>>8;
  }

  // get the next cluster value
  // cluster = 0xabcd
  // packed as:     0    |   1    |   2   |   3    |
  // word p       0   1 |  2   3 | 4   5 |  6   7 |..
  //              cd  ab|  cd  ab| cd  ab|  cd  ab|
  c = ReadOddW( fp->buffer, ((ccls & 0xFF)<<1));

  return c;

} // readFAT
```

因为 FAT 的每个扇区（从现在开始称其为页）都包含 256 个条目，很有可能在沿着簇链表前进时，或者在寻找一个空簇时（很快我们就会遇到这种情况），会不断地访问同一个页。为了不把时间浪费在不断地重复加载同一个扇区上，readFAT()函数将使用 MFILE 结构中的 fpage 字段来跟踪文件缓冲区的内容，维护最后加载的 FAT 页的索引。这需要 readDATA()函数和 readDIR()函数给予一定的配合，从而在用它们的内容（分别为文件数据和目录表条

目）覆盖缓冲区的内容时，它们能更新 fpage 索引值，把索引值置为-1 使其无效，并告知 readFAT() 函数。

15.10 关闭文件

因为目前我们只能通过已定义的 fopenM() 函数来打开文件并进行读取，因此关闭文件所需做的工作并不多。

```
unsigned fcloseM( MFILE *fp)
{
  unsigned e, r;
  r = TRUE;
  //  free up the buffer and the MFILE struct
  free( fp->buffer);
  free( fp);
  return( r);
} // fcloseM
```

15.11 fileio 模块

我们可以将目前已创建的所有函数存入名称为 fileio.c 的文件里，作为文件输入/输出函数库创建的第一步。请记得将常用的文件说明和头文件加入到 fileio.c 中：

```
/*
** fileio.c
**
** FAT16 support
*/

// standard C libraries used
#include <stdlib.h>      // NULL, malloc, free...
#include <ctype.h>       // toupper...
#include <string.h>      // memcpy...

#include <sdmmc.h>       // sd/mmc card interface
#include "fileio.h"      // file I/O routines
```

当然，还需要创建一个 fileio.h 头文件，把需要对外发布的所有定义和函数原型加入其中。

```
/*
** fileio.h
**
** FAT16 support
*/

extern char FError;                   // mailbox for error reporting

// FILEIO ERROR CODES
#define FE_IDE_ERROR           1     // IDE command execution error
#define FE_NOT_PRESENT         2     // CARD not present
#define FE_PARTITION_TYPE      3     // WRONG partition type
#define FE_INVALID_MBR         4     // MBR sector invalid signtr
#define FE_INVALID_BR          5     // Boot Record invalid signtr
#define FE_MEDIA_NOT_MNTD      6     // Media not mounted
#define FE_FILE_NOT_FOUND      7     // File not found,open for read
#define FE_INVALID_FILE        8     // File not open
#define FE_FAT_EOF             9     // attempt to read beyond EOF
#define FE_EOF                10     // Reached the end of file
#define FE_INVALID_CLUSTER    11     // Invalid cluster>maxcls
#define FE_DIR_FULL           12     // All root dir entry are taken
```

```
#define FE_MEDIA_FULL        13  // All clusters taken
#define FE_FILE_OVERWRITE    14  // A file with same name exist
#define FE_CANNOT_INIT       15  // Cannot init the CARD
#define FE_CANNOT_READ_MBR   16  // Cannot read the MBR
#define FE_MALLOC_FAILED     17  // Could not allocate memory
#define FE_INVALID_MODE      18  // Mode was not r.w.
#define FE_FIND_ERROR        19  // Failure during FILE search

typedef struct {
  LBA    fat;                    // lba of FAT
  LBA    root;                   // lba of root directory
  LBA    data;                   // lba of the data area
  unsigned short maxroot;        // max entries in root dir
  unsigned short maxcls;         // max clusters in partition
  unsigned short fatsize;        // number of sectors
  unsigned char fatcopy;         // number of copies
  unsigned char sxc;             // number sectors per cluster
  } MEDIA;

typedef struct {
  MEDIA * mda;                   // media structure pointer
  unsigned char * buffer;        // sector buffer
  unsigned short cluster;        // first cluster
  unsigned short ccls;           // current cluster in file
  unsigned short sec;            // sector in current cluster
  unsigned short pos;            // position in current sector
  unsigned short top;            // bytes in the buffer
  int    seek;                   // position in the file
  int    size;                   // file size
  unsigned short time;           // last update time
  unsigned short date;           // last update date
  char    name[11];              // file name
  char    mode;                  // mode 'r', 'w'
  unsigned short fpage;          // FAT page currently loaded
  unsigned short entry;          // entry position in cur dir
  } MFILE;

// file attributes
#define ATT_RO       1           // attribute read only
#define ATT_HIDE     2           // attribute hidden

#define ATT_SYS      4           // "     system file
#define ATT_VOL      8           // "     volume label
#define ATT_DIR      0x10        // "     sub-directory
#define ATT_ARC      0x20        // "     (to) archive
#define ATT_LFN      0x0f        // mask for Long File Name

#define FOUND        2           // directory entry match
#define NOT_FOUND    1           // directory entry not found

// macros to extract words and longs from a byte array
// watch out, a processor trap will be generated if the address
//    is not word aligned
#define ReadW( a, f) *(unsigned short*)(a+f)
#define ReadL( a, f) *(unsigned short*)(a+f)+\
      (( *(unsigned short*)(a+f+2))<<16)

// this is a "safe" versions of ReadW
```

```
//    to be used on odd address fields
#define ReadOddW( a, f)  (*(a+f)+( *(a+f+1) << 8))

// prototypes
unsigned nextFAT( MFILE * fp, unsigned n);
unsigned newFAT( MFILE * fp);

unsigned readDIR( MFILE *fp, unsigned entry);
unsigned findDIR( MFILE *fp);
unsigned newDIR ( MFILE *fp);

MEDIA * mount( void);
void    unmount( void);

MFILE *  fopenM  ( const char *name, const char *mode);
unsigned freadM  ( void * dest, unsigned count, MFILE *);
unsigned fwriteM ( void * src, unsigned count, MFILE *);
unsigned fcloseM ( MFILE *fp);

unsigned listTYPE( char *list, int max, const char *ext );
```

目前我们还没有完成所有的函数，但不必担心，在本章剩余内容的讲述中我们将逐步对其进行完善。

15.12 测试 fopenM()和 freadM()

已经很久没有建立工程了。为了验证目前已开发的这些代码的正确性，我们不得不开发出一些核心例程，没有这些核心例程，所有的应用程序都无法正确运行。现在我们已经具备了这些核心功能，因此可以首次开发一个小的测试程序，从 SD/MMC 卡读取一个创建于 FAT16 文件系统下的文件（命名为 ReadTest）。

程序思想是从 PC 机上复制一个文本文件（任何文本文件都可以）到 SD/MMC 卡上，然后用 PIC32 来读取文件，计算文件行数并将它显示到 LCD 上。

以下是保存为 ReadTest.c 的主模块：

```
/*
** ReadTest.c
**
** 07/18/07 v2.0 LDJ
** 11/23/07 v3.0 LDJ using the LCD display
*/

#include <p32xxxx.h>
#include <plib.h>
#include <explore.h>
#include <SDMMC.h>
#include <LCD.h>
#include "fileio.h"

#define B_SIZE 10
char data[ B_SIZE];

int main( void)
{
  MFILE *fs;
  unsigned r;
```

```
    int i, c;
    char s[16];

    //initializations
    initEX16();
    initLCD();                  // init LCD display

    // main loop
    while( 1)
    {
      putsLCD( "Insert card...");
      while( !getCD());         // wait for card to be inserted
      Delayms( 100);            // de-bounce
      clrLCD();

      if ( mount())
      {
        putsLCD( "mount\n");
        if ( (fs = fopenM( "Text.txt", "r")))
        {
          c = 0;
          putsLCD("Reading...");
          do{
            r = freadM( data, B_SIZE, fs);
            for( i = 0; i<r; i++)
            {
              if ( data[ i]=='\n')
              {
                c++;
                sprintf( s, "\n%d lines", c);
                putsLCD( s);
              }
            } // for i
          } while( r==B_SIZE);
          fcloseM( fs);
          homeLCD();
          putsLCD("File closed");
        }
        else
          putsLCD("File not found!");

        unmount();
      } // mounted
      else
          putsLCD("Mount Failed!");

      getKEY();
    } // loop
} // main
```

操作顺序和基本的 SD/MMC 卡访问模块测试程序的操作顺序类似，只是这次不是调用 init Media()函数，然后直接开始读取或写入 SD/MMC 卡的扇区，而是调用 mount()函数来访问存储卡上的 FAT16 文件系统。使用文件名打开数据文件，并以任意长度（B_SIZE）的数据块从文件中读取数据，扫描这些数据，找到新一行的字符并标示每一行的结束。一旦完成整个文件内容的扫描，就关闭文件并释放所有已用的存储空间。

在生成工程之前，记住将以下所有模块加入到工程中：

❑ SDMMC.c

- Fileio.c
- LCDlib.c
- Explore.c
- ReadTest.c

用检查表检查在线调试器，同时使用工程生成选项对话框（Project|Bulid Options|Project），为堆留出一些存储空间，fileio 函数才能为文件系统结构和缓冲区动态分配存储空间。即使 580B 就足够了，但还是应该给堆留出充足的回旋余地；我建议至少给堆分配 2KB。

在生成工程并对 Explore 16 演示板编程之后，就可以运行测试程序了。如果一切正确，就能在 LCD 上看到提示用户插入 SD 卡，并很快能在第二行看到计数器的更新，更新速度非常快，以至于当你看清时就已经显示最终结果了。

可以重新编译工程，并设置不同的数据缓冲区大小来运行测试程序，可以将缓冲区大小从 1B 开始设置，一直设置到 PIC32 允许的最大存储量为止。只要文件中还有数据，freadM()函数就会负责读取出用户请求的所有数据。

15.13　向文件中写入数据

革命尚未成功。只有把创建新文件的功能加入到 fileio.c 模块中以后，它才算真正完成。这需要创建一个 fwriteM()函数，完成 fopenM()函数，并大大扩展 fcloseM()函数的功能。到目前为止，如果在根目录中未找到目标文件，或者读写模式不是 r 时，fopenM()函数就会返回一个错误码。但这其实是打开新文件并写入数据所需要做的工作。检查模式参数值时，我们需要增加一个新的选择分支，把它放在未找到目标文件而又希望继续做其他处理的地方。

```
else // 11. open for 'write'
  {
    if ( r == NOT_FOUND)
    {
```

新文件需要分配一个新簇来放置数据。函数 newFAT()用于在 FAT 中搜索可用空间，用 0x0000 标示的簇被认为是可用的。搜索有可能失败，函数返回一个错误码，表示所有的存储空间已满，所有的数据簇都在使用中。如果搜索成功，那么就应该记住新簇的位置，并更新 MFILE 结构，将该簇作为新文件的第一个簇。

```
// 11.1 allocate a first cluster to it
fp->ccls = 0;              // indicate brand new file
if ( newFAT( fp) != TRUE)
{ // must be media full
  FError=FE_MEDIA_FULL;
  goto ExitOpen;
}
fp->cluster = fp->ccls;
```

接下来，需要为新文件在目录中找到一个可用的条目空间。这需要第二次搜索根目录，这次寻找的是标示为删除（0xE5 码）的第一个条目，或者是列表尾部的一个空条目（用 0x00 码标示）。

```
// 11.2 create a new entry
// search again, for an empty entry this time
if ( (r = newDIR( fp)) == FAIL)
{    // report any error
  FError  =  FE_IDE_ERROR;
  goto ExitOpen;
}
```

函数 newDIR()会寻找可用条目。和以前用过的 findDIR()函数类似，newDIR()会返回以下 3 种代码。

- FAIL，表示遇到了问题（或者存储卡被移除）。
- NOT_FOUND，表示根目录已满。
- FOUND，表示已找到可用条目。

```
// 11.3 new entry not found
if ( r == NOT_FOUND)
{
  FError=FE_DIR_FULL;
  goto ExitOpen;
}
```

在前两种情况下，必须报错并停止后续工作。但是如果找到了可用条目，就必须做大量的工作来初始化该条目。

计算出该条目在当前缓冲区中的偏移以后，需要用 MFILE 结构中的数据来填充它的一些字段。首先填充文件大小字段。

```
else // 11.4 new entry identified fp->entry filled
{
      // 11.4.1
      fp->size = 0;

      // 11.4.2 determine offset in DIR sector
      e = (fp->entry & 0xf) * DIR_ESIZE;

      // 11.4.3 init all fields to 0
      for (i=0; i<32; i++)
              fp->buffer[ e +i ] = 0;
```

时间和日期字段可以从 RTCC 模块计数器中获取，也可以采用程序的其他时间记录机制来获取。但这里为了演示的目的就只提供一个默认值即可。

```
// 11.4.4 set date and time
fp->date = 0x378A; // Dec 10th, 2007
fp->buffer[ e + DIR_CDATE] = fp->date;
fp->buffer[ e + DIR_CDATE+1] = fp->date>>8;
fp->buffer[ e + DIR_DATE] = fp->date;
fp->buffer[ e + DIR_DATE+1] = fp->date>>8;

fp->time = 0x6000; // 12:00:00 PM
fp->buffer[ e + DIR_CTIME] = fp->time;
fp->buffer[ e + DIR_CTIME+1] = fp->time>>8;
fp->buffer[ e + DIR_TIME] = fp->time+1;
fp->buffer[ e + DIR_TIME+1] = fp->time>>8;
```

接下来填充目录条目中的文件的第一个簇编号、文件名以及属性（默认）。

```
// 11.4.5 set first cluster
fp->buffer[ e + DIR_CLST] = fp->cluster;
fp->buffer[ e + DIR_CLST+1] = (fp->cluster>>8);

// 11.4.6 set name
for ( i = 0; i<DIR_ATTRIB; i++)
     fp->buffer[ e + i] = fp->name[i];

// 11.4.7 set attrib
fp->buffer[ e + DIR_ATTRIB] = ATT_ARC;
```

```
            // 11.4.8 update the directory sector;
            if ( !writeDIR( fp, fp->entry))
            {
                FError=FE_IDE_ERROR;
                goto ExitOpen;
            }
        } // new entry
    } // not found
```

回到第一次搜索根目录得到的结果处。如果确实找到了具有相同文件名的文件，那么必须报错。

```
    else // file exist already, report error
    {
        FError = FE_FILE_OVERWRITE;
        goto ExitOpen;
    }
```

还有一种替代方案，就是先删除当前条目，释放所有使用的簇，然后从头开始。当然，把遇到的问题当成错误进行报告是目前最简单的处理办法。

以上就是 fopenM() 函数需要修改的内容。现在可以开始编写新的 fwriteM() 函数了，此处再次模仿标准 C 库函数的命名方法。

```
    unsigned fwriteM( void *src, unsigned count, MFILE * fp)
    // src      points to source data (buffer)
    // count    number of bytes to write
    // returns  number of bytes actually written
    {
        MEDIA *mda = fp->mda;
        unsigned len, size = count;

        // 1. check if file is open
        if ( fp->mode != 'w')
        {   // file not valid or not open for writing
            FError = FE_INVALID_FILE;
            return FAIL;
        }
```

传递给该函数的参数和 freadM() 函数中的参数相同。结构体 MFILE 作为参数之一，和 freadM() 函数中的定义相同。fwriteM() 函数也可以算作是对 MFILE 结构体所做的第一次全面测试。测试结果将决定通过调用 fopenM() 函数而填充的 MFILE 结构的内容是否值得信任。

函数的核心部分也是一个循环：

```
    // 2. loop writing count bytes
    while ( count>0)
    {
```

我们希望通过 string.h 函数库里的 memcpy() 函数，每次能传输尽可能多的数据。

```
        // 2.1 copy as many bytes at a time as possible
        if ( fp->pos+count < 512)
          len = count;
        else
          len = 512- fp->pos ;

    memcpy( fp->buffer+ fp->pos, src, len);
```

在往缓冲区添加数据和增加文件大小时，需要更新一些指针和计数器的内容，来跟踪当前位置。

```
// 2.2 update all pointers and counters
fp->pos+=len;        // advance buffer position
fp->seek+=len;       // count the added bytes
count-=len;          // update the counter
src+=len;            // advance the source pointer

// 2.3 update the file size too
if (fp->seek > fp->size)
  fp->size = fp->seek;
```

一旦缓冲区填满，就需要将数据传输到当前分配簇的某个扇区中：

```
// 2.4 if buffer full, write current buffer to current
sector
if (fp->pos == 512)
{
  // 2.4.1 write buffer full of data
  if ( !writeDATA( fp))
      return FAIL;
```

注意，这里如果产生错误，就有可能是致命的。如果所有的数据传输都失败，那么就会返回代码 FAIL，其值为 0。这也就表示所有写入存储设备的数据其实已经丢失了。

如果所有的步骤都已正确执行，那么就该递增扇区指针了，然而如果当前簇的所有扇区都已经写入完毕，那就得考虑再次调用 newFAT() 函数来分配一个新簇。

```
// 2.4.2 advance to next sector in cluster
fp->pos = 0;
fp->sec++;

// 2.4.3 get a new cluster if necessary
if ( fp->sec == mda->sxc)
{
  fp->sec = 0;
  if ( newFAT( fp)== FAIL)
      return FAIL;
    }
  } // store sector
} // while count
```

简而言之，在开发 newFAT() 函数时，必须保证该函数在给文件分配新簇的同时，能够准确维护 FAT 中的簇链表。

```
  // 3. number of bytes actually written
  return size-count;
} // fwriteM
```

fwriteM() 函数到此已完成，可以把退出循环时已写入的字节数返回了。

15.14 关闭文件（续）

关闭一个用于读取的文件很简单，只需释放堆上的一些空间即可。但是如果关闭一个用于写入的文件，那就还需要执行很多额外的收尾工作。

因此需要开发一个新的 fcloseM() 函数，该函数从检查 mode 字段开始。

```
unsigned fcloseM( MFILE *fp)
{
  unsigned e, r;
  r = FAIL;

  // 1. check if it was open for write
  if ( fp->mode == 'w')
  {
```

实际上在关闭文件时，可能仍然有些数据残留在缓冲区里需要写入存储设备，尽管这些数据并不能填满整个扇区。

```
// 1.1 if the current buffer contains data, flush it
if ( fp->pos >0)
{
  if ( !writeDATA( fp))
      goto ExitClose;
}
```

这里发生的错误也可能是致命的，意味着所有的文件数据都丢失了，因为 fcloseM() 函数不能正确完成了。

必须检索到正确的根目录扇区，并正确计算缓冲区内目录条目的偏移。

```
// 1.2    finally update the dir entry,
// 1.2.1   retrive the dir sector
if ( !readDIR( fp, fp->entry))
  goto ExitClose;

// 1.2.2 determine position in DIR sector
e = (fp->entry & 0xf) * DIR_ESIZE;
```

接下来用实际文件大小来更新根目录中的文件条目（之前初始化为 0）。

```
// 1.2.3 update file size
fp->buffer[ e + DIR_SIZE]   = fp->size;
fp->buffer[ e + DIR_SIZE+1]= fp->size>>8;
fp->buffer[ e + DIR_SIZE+2]= fp->size>>16;
fp->buffer[ e + DIR_SIZE+3]= fp->size>>24;
```

最后，把包含该条目的整个根目录扇区写回到存储设备上去。

```
// 1.2.4 update the directory sector;
if ( !writeDIR( fp, fp->entry))
  goto ExitClose;
} // write
```

如果一切正常，那么就算完成了 fcloseM() 函数，可以释放存储器空间了。

```
    // 2. exit with success
    r = TRUE;

ExitClose:
    // 3. free up the buffer and the MFILE struct
    free( fp->buffer);
    free( fp);

  return( r);

} // fcloseM
```

15.15 辅助函数

在完成 fopenM()、fcloseM()以及创建新的 fwriteM()函数时，我们已经使用了一些底层函数来执行一些重要的重复工作。

我们从 newDIR()函数开始，该函数用于在根目录中寻找可用空间来创建一个新文件。它和 findDIR()函数的相似性是显而易见的，只是执行的任务不同。

```
unsigned newDIR( MFILE *fp)
// fp      file structure
// return  found/fail, fp->entry filled
{
  unsigned eCount;              // current entry counter
  unsigned e;                   // current entry offset
  int a;
  MEDIA *mda = fp->mda;

  // 1. start from the first entry
  eCount = 0;
  // load the first sector of root
  if ( !readDIR( fp, eCount))
    return FAIL;

  // 2. loop until you reach the end or find the file
  while ( 1)
  {
  // 2.0 determine the offset in current buffer
    e = (eCount&0xf) * DIR_ESIZE;

  // 2.1 read the first char of the file name
    a = fp->buffer[ e + DIR_NAME];

  // 2.2 terminate if it is empty (end of the list)or deleted
    if (( a == DIR_EMPTY) ||( a == DIR_DEL))
    {
      fp->entry = eCount;
      return FOUND;
    } // empty or deleted entry found

  // 2.3 get the next entry
    eCount++;
    if ( (eCount & 0xf) == 0)
    { // load a new sector from the root
      if ( !readDIR( fp, eCount))
          return FAIL;
    }

  // 2.4 exit the loop if reached the end or error
    if ( eCount > mda->maxroot)
      return NOT_FOUND;                // last entry reached
  }// while

  return FAIL;
} // newDIR
```

函数 newFAT()则用于寻找一个可用的簇，分配给新的数据块或新的文件：

```
unsigned newFAT( MFILE * fp)
// fp    file structure
// fp->ccls     ==0 if first cluster to be allocated
//              !=0 if additional cluster
// return   TRUE/FAIL
// fp->ccls new cluster number
{
  unsigned i, c = fp->ccls;

  // sequentially scan through FAT
  do {
    c++;    // check next cluster in FAT
    // check if reached last cluster in FAT,
    // re-start from top
    if ( c >= fp->mda->maxcls)
      c = 0;

    // check if full circle done, media full
    if ( c == fp->ccls)
    {
      FError = FE_MEDIA_FULL;
      return FAIL;
    }

   // look at its value
   i = readFAT( fp, c);

  } while ( i!=0);       // scanning for an empty cluster

  // mark the cluster as taken, and last in chain
  writeFAT( fp, c, FAT_EOF);

  // if not first cluster, link current cluster to new one
  if ( fp->ccls >0)
    writeFAT( fp, fp->ccls, c);

  // update the MFILE structure
  fp->ccls = c;

  // invalidate the FAT cache
  //   (since it will soon be overwritten with data)
  fp->fpage = -1;

  return TRUE;
} // newFAT
```

在分配第一个簇之后的新簇时，newFAT()函数继续保持这些簇在链表中的链接，但是会把它们都标识为已用。在这个过程里，newFAT()函数使用了另一个辅助函数，即writeFAT()函数。该函数用于更新FAT以及它所有副本的内容。

```
unsigned writeFAT( MFILE *fp, unsigned cls, unsigned v)
// fp       MFILE structure
// cls      current cluster
// v        next value
// return   TRUE if successful, or FAIL
{
  unsigned p;
  LBA l;

  // get address of current cluster in fat
  p = cls * 2; // always even
```

```
    // cluster = 0xabcd
    // packed as:      0  | 1     | 2     | 3     |
    // word p        1  2 | 3  4  | 4  5  | 6  7  |..
    //               cd ab| cd ab| cd ab| cd ab|
    // load the fat sector containing the cluster
    l = fp->mda->fat + (p >> 9 );
    p &= 0x1fe;
    if ( !readSECTOR( l, fp->buffer))
      return FAIL;
    // get the next cluster value
    fp->buffer[ p] = v;          // lsb
    fp->buffer[ p+1] = (v>>8); // msb
    // update all FAT copies
    for ( i=0; i<fp->mda->fatcopy; i++, l += fp->mda->fatsize)
      if ( !writeSECTOR( l, fp->buffer))
          return FAIL;
    return TRUE;
  } // writeFAT
```

最后，fwriteM()函数和fcloseM()函数都使用了writeDATA()函数把数据写入到实际的存储设备扇区中，并基于当前簇编号来计算扇区的地址。

```
unsigned writeDATA( MFILE *fp)
{
  LBA l;
  // calculate lba of cluster/sector
  l=fp->mda->data+(LBA)(fp->ccls-2) * fp->mda->sxc+fp->sec;
  return ( writeSECTOR( l, fp->buffer));
} // writeDATA
```

15.16 测试完整的 fileio 模块

现在来测试我们刚完成的整个 fileio.c 模块的功能。这次的任务是在挂载文件系统以后，打开一个源文件（可以是任何文件），把它的内容复制到当场创建的新“目标”文件中。以下就是 WriteTest.c 主文件的代码：

```
/*
**WriteTest.c
**
*/
#include <p32xxxx.h>
#include <explore.h>
#include <LCD.h>
#include <SDMMC.h>
#include "fileio.h"
#define B_SIZE 100
char data[B_SIZE];
int main( void)
{
  MFILE *fs, *fd;
  unsigned c, i, p, r;
  char s[32];
```

```
    //initializations
    initEX16();
    initLCD();                    //init LCD display
    putsLCD( "Insert card...\n");
    while( !getCD());             // wait for card to be inserted
    Delayms( 100);                // wait for card to power up
    if ( mount())
    {
      clrLCD();
      if ( (fs = fopenM( "source.txt", "r")))
      {
        if ( (fd = fopenM( "dest.txt", "w")))
        {
          c = 0;                  // init byte counter
          p = 0;                  // init progress index
          i = fs->size/16;        // progress bar increment
        putsLCD("Copying\n");
        do{
          // copy data
          r = freadM( data, B_SIZE, fs);
          r = fwriteM( data, r, fd);
          // update progress bar
          c += r;
          while (p < c/i)
          {
            p++;
            putLCD( 0xff);   // add one bar
          }
        } while( r == B_SIZE);
        r = fcloseM( fd);
        if ( r == TRUE)
        {
          clrLCD();
          sprintf( s, "Copied \n%d bytes", c);
          putsLCD( s);
        } // close dest
        else
          putsLCD("ER:closing dest");
        } // open dest
        else
          putsLCD("ER:creating file");
        fcloseM( fs);
      } // open source
      else
        putsLCD("ER:open source");
      unmount();
    } // mount
    else
      putsLCD("ER:mount failed");
    // main loop
    while( 1);
} // main
```

确保把上述代码中的源文件名（SOURCE.TXT）替换成实验时将实际复制到存储卡上的文件名。

在创建新工程以后（这次将工程命名为 WriteTest），需要把所有必需的模块加入到工程窗口中，包括：

- SDMMC.c
- fileio.c
- explore.c
- LCDlib.c
- WriteTest.c

请再次根据 New Project（新工程）检查表以及 in-circuit debugger setup（在线调试器设置）检查表来进行检查，这次要记住给堆分配更多空间，从而能够为两个 MFILE 结构动态分配两个缓冲区。



> **注解**　一旦给全局变量和栈留够了空间，就没有理由在堆空间分配上变得吝啬了。分配尽量大的堆空间，能够让 malloc() 和 free() 函数更优化地使用所有可用的存储空间。

生成工程，对 Explorer 16 演示板进行编程，然后运行测试程序。在给出提示信息时插入 SD 卡，如果一切运行正确，经过不到一秒钟（时间长短与所选源文件大小相关），就能看到进度条逐渐充满 LCD 的第二行。当复制完成时，LCD 上会出现与下列信息类似的一条消息：

```
Copied
1806 bytes
```

实际的字节数会反映出源文件的大小。这时把 SD/MMC 卡上的文件再传回 PC 机，就可以验证新文件已创建（参见图 15-10）。

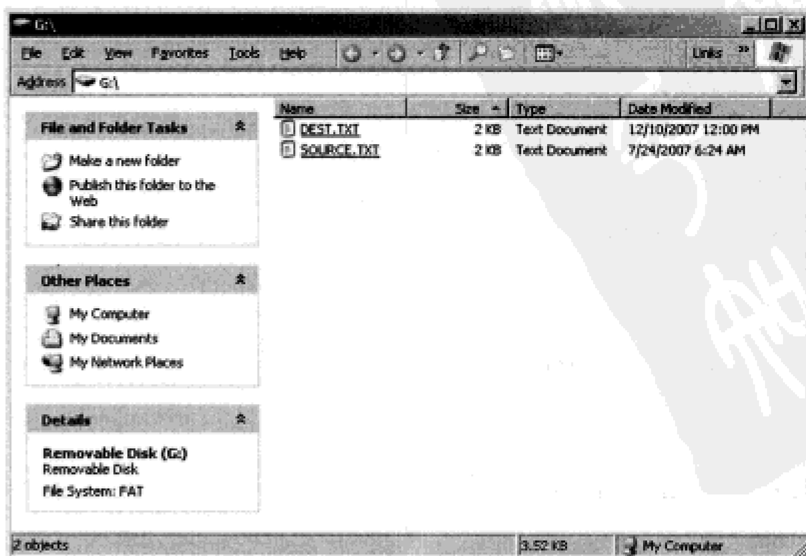


图 15-10　Windows 资源浏览器截图

它的大小和内容与源文件完全相同，只有日期和时间表示出它是在 fopenM() 函数中设置的。需要注意的是，如果试图再次运行测试程序，那势必会报错。

```
ER:creating file
```

在开发 fopenM() 函数时已经讨论过了，在创建新文件（即为写入而打开一个文件）而又发现同名文件已存在时，选择的处理措施是报错。

可以选择不同的数据缓冲区大小，从 1B 开始设置，一直到 PIC32 允许使用的最大容量为止，然后重新编译工程并运行程序。freadM() 函数和 fwriteM() 函数都会负责读写请求的所有数据，不过完成操作的时间会有细微变化。

15.17 代码体积

WriteTest 工程产生的代码体积与前一章里测试过的简单 SDMMC.c 模块的代码体积相比，显然要大很多（参加图 15-11）。

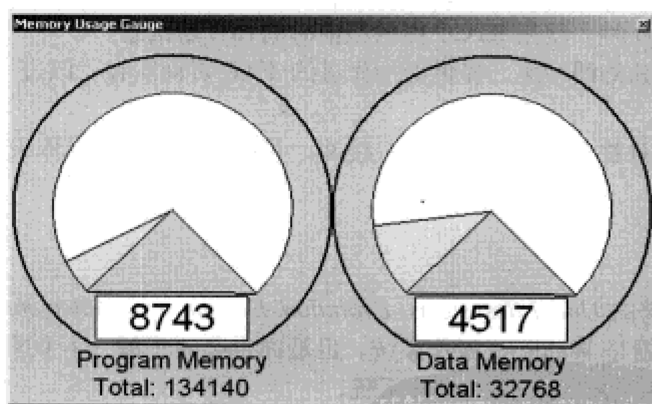


图 15-11 存储器用量

当然，如果没有开启编译器优化选项，代码体积加起来也只有 8 743 个字。这仅仅是 PIC32MX360 上可用存储空间总数的 6%。我认为以这么小的代价换取这么多的功能是非常值得的！

15.18 小结

在本章中我们学习了 FAT16 文件系统的基本知识，并开发了一个小型接口模块，使 PIC32 单片机可以从通用大容量存储设备读写文件数据。使用在前一章里为底层接口开发的 SDMMC.c 模块，我们为 SD/MMC 存储卡开发了一个基本的文件 I/O 接口。

现在就可以在 PIC32 应用程序中与几乎所有能访问 SD/MMC 存储卡的计算机系统共享数据了，包括 PDA、笔记本和台式机，运行 DOS、Windows、Linux 系统的机器以及运行 OS-X 的苹果电脑。

15.19 提示与技巧

嵌入式控制应用工程师经常问我的一个问题是："我如何和'拇指盘'（有时候被称为 USB 记忆棒）、USB 存储器连接，从而让我的嵌入式应用和 PC 机共享并互传数据？"

我的答案很简单："哪怕可以也别这样做。"解决方案则是："用 SD 卡就行了啊！"下面我

就来说明为什么。正如在本章和前一章中所看到的，读写 SD 卡（也包括 miniSD 和 microSD 卡）真地很简单，只需要一个 SPI 端口和一小段代码。

而且，从用户的角度来看，USB 接口的确具有吸引力，外观也很简单，但是读写 USB 设备对于当前的嵌入式控制应用来说，相当复杂并且价格昂贵。首先，必须用相对复杂的 USB 总线接口替代简单的 SPI 接口。然后，不仅还需要一个标准的 USB 接口，另外还需要一个主 USB 接口并开发相应的软件代码。

在写这本书时，PIC32 开发商已经宣布在后续版本中提供集成的主 USB 接口了，但是从支持完整软栈所需的 Flash 和 RAM 用量来看，价格不会很便宜。和我们今天研究的基本 SD/MMC 存储卡读写方案相比，其所需容量必定是几何级数倍的增长，而且处理起来也会复杂得多。

15.20 练习

（1）回顾 PIC32 工具包中提供的 FAT16 支持库。现在你已经掌握了这些工具，可以更好地理解代码，更自信地使用一些更先进的特性。

（2）在写入新文件时，使用 RTCC 提供当前时间和日期信息。

（3）利用一个单独的缓冲区，提供更加先进的 FAT 页面缓存，以进一步改善读/写性能，并对其效果进行评估。

（4）修改程序，对整个簇的内容都进行缓冲，并执行多块读写操作来优化 SD 卡的底层性能，对效果进行评估。

15.21 参考书

Steve D. Pate 所著的 *Unix Filesystems: Evolution, Design, and Implementation*。和个人计算机共享文件首先关注的就是 Windows 操作系统，但是你也必须了解一下 UNIX（和 Linux），从而能为关键任务的数据存储找到正确的文件系统。

15.22 链接

www.tldp.org/LDP/tlk/tlk-title.html。David A Rusling 所写的 *The Linux Kernel*。本书描述了 Linux 及其文件系统的内部工作机制。

http://en.wikipedia.org/wiki/File_Allocation_Table。这是维基百科提供的另一个非常棒的页面，描述了 FAT 技术的历史以及很多相关知识。

http://en.wikipedia.org/wiki/List_of_file_systems。列出了使用中的所有主要计算机文件系统并进行分类。

http://en.wikipedia.org/wiki/ISO-9660。想知道文件是如何写入光盘的吗？答案就是 ISO-9660 文件系统。

第 16 章　音乐播放器

16.1　计划

以模拟信号存储音乐、家庭音响是一整套昂贵的电子设备的时代已经一去不复返了。从大约 20 年前的音乐 CD 开始，一直到如今的 iPod 和 MP3 播放器，音乐早已开始以数字的形式存储和消费了。对于消费者和嵌入式应用而言，数字音频是一种可行而又便宜的娱乐方式，同时还可以作为和用户进行交流的手段。

在本章中，我们将研究如何使用 PIC32 的输出比较模块产生音频信号。在 PWM 模式下，结合比较高级的低通滤波器的使用，输出比较模块可以作为有效的 DAC，用来产生模拟输出信号。采用 20Hz ~ 20kHz 之间的、能被人耳识别的频率把模拟信号调制好以后，我们就能听到声音了！

16.2　准备

除了 MPLAB IDE、MPLAB C32 编译器以及 MPLAB SIM 仿真器在内的这些常见软件工具之外，本章还需要用到 Explorer 16 演示板和用户自行选择的在线调试器。你还需要准备一个电烙铁和一些元器件，从而可以通过原型板区或者小型扩展板来扩展 Explorer 16 演示板的功能。你还可以访问本书配套网站（www.exploringPIC32.com）来获取有关扩展板的更多信息，从而更好地完成本章的实验。

16.3　探索

PWM 信号的工作方式非常简单。以定时器及其周期寄存器产生的规则时间间隔来产生脉冲，尽管脉冲宽度（T_{on}）不是固定的，但是它是可编程的，可以在时间间隔的 0% ~ 100%之间变化。脉冲宽度（T_{on}）和信号周期（T）之间的比率称为占空比（duty cycle），如图 16-1 所示。

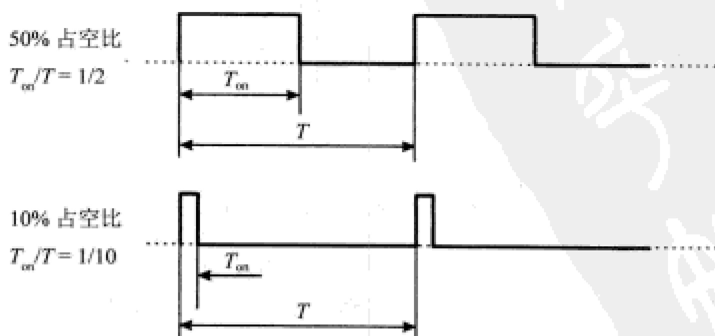


图 16-1　不同占空比的 PWM 信号示例

占空比有两个极端情况：0%和 100%。第一种情况对应于总是关闭的信号。第二种情况则对应于总是开启的信号。介于这两者之间的占空比的数量，通常是一个相对较小的有限数，用以 2 为底的对数形式来表示，称为 PWM 的分辨率。例如，如果有 256 种可能的脉冲宽度，那么我们说这个 PWM 信号具有 8 位的分辨率。

如果给频谱分析仪输入一个具有固定占空比的理想 PWM 信号，用以分析其组成，就会发现它包含 3 个部分（参见图 16-2）。

- 一个直流分量，振幅和占空比成正比。
- 基频下的正弦曲线（f=1/T）。
- 接下来是无穷多个谐波，其频率是基频的倍数（2f，3f，4f，5f，6f，等等）。

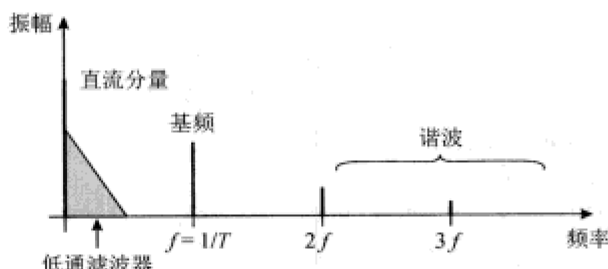


图 16-2　PWM 信号的频谱

因此，如果可以附加一个理想的低通滤波器到 PWM 信号产生器的输出上，用以移除所有基频以上的谐波，就可以获取一个干净的 DC 模拟信号，其振幅和占空比成正比。

当然，这样的理想滤波器是不存在的，但是可以采用差不多接近的高级滤波器来移除尽可能多的不需要的频率分量（参见图 16-3）。这个滤波器可以是简单的单无源 R/C 电路（一阶低通滤波器），也可以是 N 个有源滤波器（2xN 阶低通滤波器）。

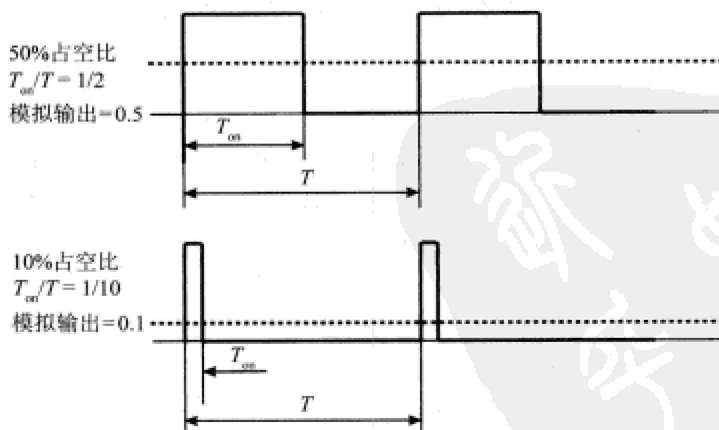


图 16-3　PWM 的模拟信号输出以及理想的低通滤波器电路

如果想产生一个音频信号，并选择合适的 PWM 频率，那么可以利用人耳的天然特性。人耳天生就可以当成附加的滤波器来使用，它能忽略频率在 20Hz~20kHz 之外的所有信号。另外，音频输出信号会再次输入给音频放大器，而大部分音频放大器在其输入部分都包含一个类似的滤波器。换句话说，如果 PWM 信号是在 20kHz 或高于 20kHz 的频率下工作，那么以上两种措施就可以处理这种情况，也就意味着只需要使用更简单和更便宜的滤波器电路来处理其他情况了。

很明显，在每个 PWM 周期（T）内，不能多次改变占空比，PWM 频率越高，改变输出模拟信号的速度就越快，因此能产生的音频信号的频率也越高。

从实用的角度来讲，这意味着 PWM 能产生的音频信号的最高频率只是 PWM 频率的一半。那么比如说，一个 20kHz 的 PWM 电路只能重复产生最高频率为 10kHz 的音频信号，而要覆盖整个能被人耳听到的频谱范围，基本周期就必须至少是 40kHz。现在你理解了为何音乐 CD 的编码速率是每秒 44 100 个样本，这并非巧合。

16.4 OC PWM 模式

在前文中，我们使用了 PIC32 的输出比较模块来产生精确的时间间隔（从而获取产生复合视频输出信号所需的水平同步信号）。这次我们在 PWM 模式下使用 OC 模块来产生具有所需占空比的连续脉冲流。

为了初始化 OC 模块来产生一个 PWM 信号，我们需要做的工作就是将 OCxCON 控制寄存器中的 3 个 OCM 位设置为基本的 PWM 配置值 0x110（参见图 16-4）。也可以使用第二种 PWM 模式（0x111），但是故障输入引脚对我们并没有什么用，通常只有作为保护机制的应用才会使用到（电机控制/电力转换）。接下来我们需要选择定时器来产生基本的 PWM 周期。选择限制在定时器 2 或定时器 3 上，但是因为在视频工程中已经用过了定时器 3，所以这次选择定时器 2（如图 16-5 所示）。

U-0	U-0	U-0	U-0	U-0	U-0	U-0	U-0
—	—	—	—	—	—	—	—
位 31							位 24

U-0	U-0	U-0	U-0	U-0	U-0	U-0	U-0
—	—	—	—	—	—	—	—
位 23							位 16

R/W-0	R/W-0	R/W-0	U-0	U-0	U-0	U-0	U-0
ON	FRZ	SIDL	—	—	—	—	—
位 15							位 8

U-0	U-0	R/W-0	R-0	R/W-0	R/W-0	R/W-0	R/W-0
—	—	OC32	OCFLT	OCTSEL	OCM<2:0>		
位 7							位 0

图 16-4 输出比较模块的主控制寄存器 OCxCON

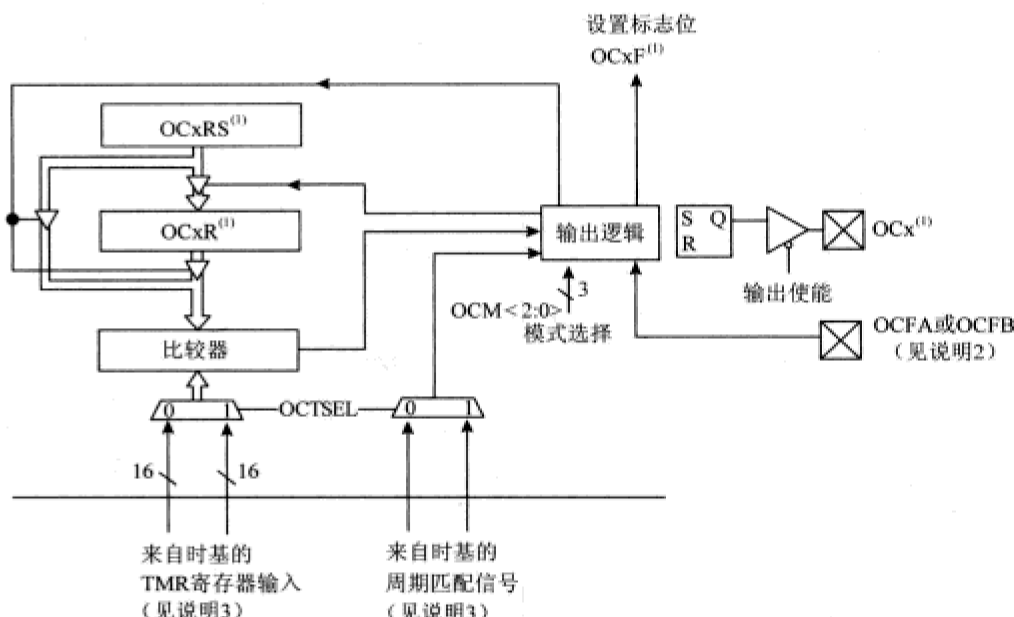
我们需要产生频率至少为 44.1kHz 的 PWM 周期，并假设外围设备时钟是 36MHz（这是使用 Explorer 16 演示板的标准配置），那么就可以计算出定时器 2（T2CON）及其周期寄存器（PR2）的优化配置值。把预分频器比率设置为 1∶1，在产生精确的 44.1kHz 的 PWM 频率时，每个周期就能够获得 816 个时钟节拍。这个值也表示了输出比较模块占空比的最大分辨率。

因为占空比有 816 个可能值，那么可以说分辨率介于 9 和 10 位之间，因为可取值大于 512（2^9），又小于 1024（2^{10}）。

把频率降到 20kHz 就可以多获得 1 位的分辨率（介于 10 和 11 之间），但是这也意味着输出频率的范围限制为最大 10kHz，对于人耳来说会有细微的但又能察觉到的差别。

配置好所选定时器之后，必须把占空比的值写入寄存器 OCxR 和寄存器 OCxRS 中，然后才能配置 OCxCON 寄存器。在 PWM 模式下，这两个寄存器工作在主/从配置方式下。一旦 PWM 模块启动（在 OCxCON 寄存器中写入模式位之后），就能够仅通过写入 OCxRS 寄存器（从）来改变占空比。OCxR 寄存器（主）会在每个新 PWM 周期开始时，从 OCxRS 寄存器中复制新值

来进行更新，从而避免毛刺，并留出整个周期（T）的时间来准备下一个占空比值。



说明：1. 图中的x表示与从1到5的不同输出比较通道相关联的不同寄存器

2. OCFA引脚控制OC1~OC3通道。OCFB引脚控制OC4~OC5通道

3. 每个输出比较通道可以在2个可选的16位定时器间选择其一，也可以使用单个32位定时器

图 16-5　输出比较模块框图

以下是一个简单的 OC1 模块初始化例程示例：

```
void initDA( int samplerate)
{
  // init OC1 module
  OpenOC1( OC_ON | OC_TIMER2_SRC | OC_PWM_FAULT_PIN_DISABLE, 0, 0);

  // init Timer2 mode and period (PR2)
  OpenTimer2( T2_ON | T2_PS_1_1 | T2_SOURCE_INT,
      FPB/samplerate);
  PR2 = FPB/samplerate-1;
  mT2SetIntPriority( 4);
  mT2ClearIntFlag();
  mT2IntEnable( 1);
} // initDA
```

注意，我们也利用初始化的机会使能了定时器中断，这样在每个周期开始时就能得到提示，并决定如何以及是否把下一个占空比值写入 OC1RS（或者使用 SetDCOC1PWM()函数）。

16.5　把 PWM 作为 D/A 转换器进行测试

要想用 Explorer 16 演示板做实验，必须先在原型板区加入一些分散的器件。用一个 1kΩ 的电阻和一个 100nF 的电容，就可以构成一个最简单的低通滤波器（截止频率为 1.5kHz 的一阶低通滤波器）。把这两个器件串联起来，并连接到 OC1 模块的输出引脚上，即 PORTD 的 0 引脚。图 16-6 给出了原理图。

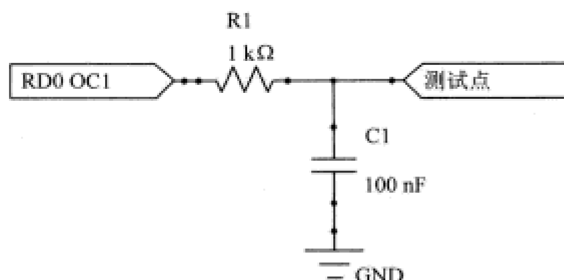


图 16-6 使用 PWM 信号生成模拟输出

还需要加入几行代码来完成本次小实验：

```
void __ISR( _TIMER_2_VECTOR, ipl4) T2Interrupt( void)
{
  // clear interrupt flag and exit
  mT2ClearIntFlag();
} // T2 Interrupt

main( void)
{
  initEX16();          // init and enable vectored interrupts
  initDA( 44100);      // init the PWM for 44.1kHz
  SetDCOC1PWM( PR2/2);

  // main loop
  while( 1);

}// main
```

加入常用的头文件，然后保存为一个新文件，命名为 TestDA.c。接下来就可以快速创建一个只包含这个文件的工程（称为 Audio）。生成工程，用在线调试器对 Explorer 16 演示板进行编程。如果有仪表或者示波器探头，把它连接到图 16-6 中的测试点上，然后运行程序验证输出电平的平均值。

仪表的探针（或者示波器的轨迹）会摆动，指示电压平均值为 1.5V；这是 Explorer 16 演示板数字 I/O 引脚常规输出电压的 50%。这和初始化例程中设置的占空比值是一致的，在初始化例程中，将占空比设置为 PWM 周期的一半（PR2/2）。如果有示波器，也可以直接把探头连接到 R1 电阻的另一端（直接连到 OC1 模块的输出引脚上），并验证会有频率刚好为 44.1kHz 的方波出现在屏幕上，并且占空比为 50%（如图 16-7 所示）。

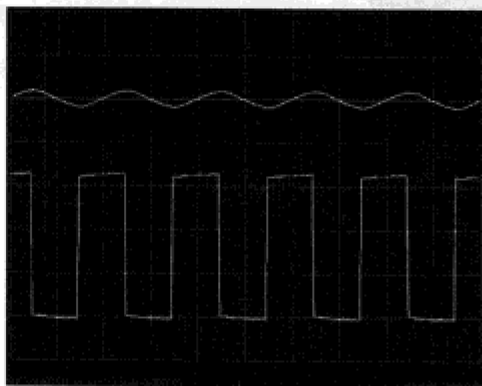


图 16-7 OC1 输出（下方）和滤波器（上方）的截图

现在可以修改初始化例程，测试 0 和 PR2 之间的其他占空比值，验证电路的响应以及占空比值与 0 到 3V 间输出电压的比例。

16.6 产生模拟波形

通过 OC1 模块的使用，我们已经跨出由 0 和 1 组成的数字世界的边界，来到了可以生成介于 0 和 3V 之间的很多电平值的模拟世界。

现在开始逐个周期地改变占空比的值，从而生成各种类型和形状的波形。首先修改工程，加入一些代码到目前暂时为空的中断服务例程：

```
void __ISR( _TIMER_2_VECTOR, ipl4) T2Interrupt( void)
{
  OC1RS = (count < 22) ? PR2 : 0;
  count++;
  if ( count >= 44)
    count = 0;

  // clear interrupt flag and exit
  mT2ClearIntFlag();
} // T2 Interrupt
```

需要把 count 声明为全局整型变量，并将其初始化为 0。

将新的代码保存为名为 TestDA2.c 的文件，替换工程中的主文件，重新生成工程并用 Explorer 16 演示板进行测试。

每 20 个 PWM 周期，滤波器输出就会在 3V（100%）和 0V（0%）之间改变一次，在示波器上产生一个频率近似 1kHz（44.1kHz/44）的方波，如图 16-8 所示。



图 16-8 TestDA2 的输出，1kHz 的方波

通过下述算法还可以产生更有趣的波形：

```
void __ISR( _TIMER_2_VECTOR, ipl4) T2Interrupt( void)
{
  OC1RS = count*PR2/44;
  count++;
  if ( count >= 44)
    count = 0;
```

```
    // clear interrupt flag and exit
    mT2ClearIntFlag();
} // T2 Interrupt
```

这将产生一个峰值振幅大概为 3V 的三角形（锯齿形）波形，占空比分 40 步在 0%到 100%间逐渐变化（每一步变化 2.5%），然后突然下降为 0。这个过程会一直重复。重复的频率也约为 1kHz（如图 16-9 所示）。

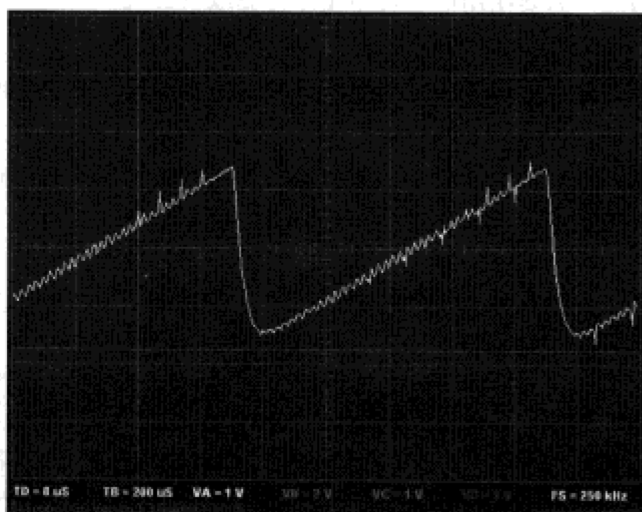


图 16-9　TestDA3 的输出，1kHz 的三角波

把新代码保存为 TestDA3.c，替换工程中的主文件并重新生成工程。

如果把这两个示例的输出送入音频放大器，听到的声音都不好听，尽管这两个输出都具有可识别的（基础的）1kHz 的高音调，但是掺杂在其中的大量谐波是能被人耳听到的，因而听起来总是带有令人不快的嗡嗡声。

为了生成干净的声音，需要产生一个纯粹的正弦波。以下给出的中断服务例程可以做到这一点。它可以生成一个频率为 441Hz 的完美的正弦波形；从音乐的角度来讲，这段波形将非常接近 A4 音调（对于不是用现代 Boethian 符号，而是用老的 Do-Re-Mi-Fa-Sol-La-Si 来学习音乐知识的我们来说，就是 La 音）。

```
void __ISR( _TIMER_2_VECTOR, ipl4) T2Interrupt( void)
{
    // compute the new sample for the next cycle
    OC1RS = PR2/2 + PR2/2 * sin(count* 2*M_PI/100);
    count++;

    // clear interrupt flag and exit
    mT2ClearIntFlag();
} // T2 Interrupt
```

可是，PIC32 以及 MPLAB C32 编译器的数学库函数速度太快，因此我们没法使用（浮点）sin()函数，也无法在 440Hz 的速率下执行乘法和加法操作来计算新的占空比值。定时器 2 可以每 22μs 中断一次，对于执行如此复杂的浮点计算来说时间太短了。因此，中断服务例程不能完全执行完毕，只能生成频率为所需频率一半（或更少）的正弦输出（低了 8 度音）。而为了实

时性能考虑，我们需要把正弦函数的值预先制成表格，从而把计算量减到最少，最好只对整数进行操作。以下是一个使用常量表的示例，常量表中包含了存储在 PIC32 的 Flash 程序存储器中的预计算值：

```
const short Table[ 100]={
// insert comma separated values here...
};
```

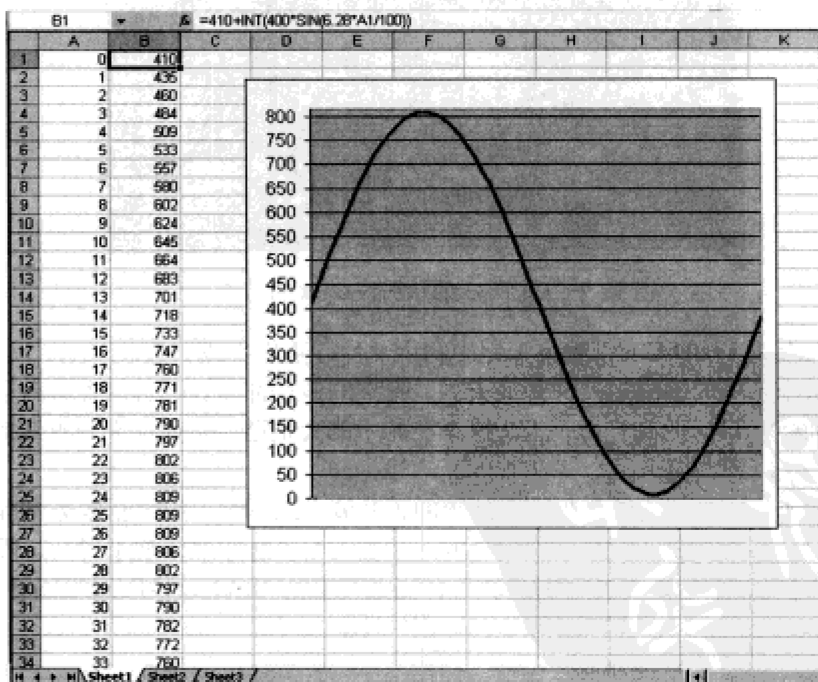
为了获取表里面的值，需要使用电子表格程序计算以下公式：

```
= offset + INT( amplitude * SIN( ROW * 6.28/ PERIOD))
```

代入周期值为 100 个样本（441Hz），偏移量为 410，振幅为 400，得到：

```
=410 + INT( 400*SIN(6.28*A1/100))
```

我们先用计数器的值填充数据表的第一列（A），然后对第二列（B）的前 100 行进行公式复制，并把输出格式调整为不显示小数位（如图 16-10 所示）。



	A	B
1	0	410
2	1	436
3	2	460
4	3	484
5	4	509
6	5	533
7	6	557
8	7	580
9	8	602
10	9	624
11	10	645
12	11	664
13	12	683
14	13	701
15	14	718
16	15	733
17	16	747
18	17	760
19	18	771
20	19	781
21	20	790
22	21	797
23	22	802
24	23	806
25	24	809
26	25	809
27	26	809
28	27	806
29	28	802
30	29	797
31	30	790
32	31	782
33	32	772
34	33	760

图 16-10 计算 100 个点的正弦值的电子表格

选择 B 列的前 100 个单元，直接把它们复制到 MPLAB 编辑器里。在每行末尾添加分号，并在表的末尾添加花括号：

```
const short Table[ 100]={
// insert comma separated values here...
410,
435,
460,
484,
509,
533,
```

```
...
383};
```

新的中断服务例程只需要让 OC1RS 的值在表格中的每个元素间轮转即可：

```
void __ISR( _TIMER_2_VECTOR, ipl4) T2Interrupt( void)
{
  OC1RS = Table[ count++];
  if ( count >= 100)
    count = 0;
  // clear interrupt flag and exit
  mT2ClearIntFlag();
} // T2 Interrupt
```

这次我们可以很容易地生成想听到的音调了，并且在定时器 2 的中断调用之间，还有足够多的时间来执行其他任务。

把新文件保存为 TestDA4.c 并替换工程中的主文件。生成工程，对 Explorer 16 演示板进行编程，并观察输出结果（如图 16-11 所示）。

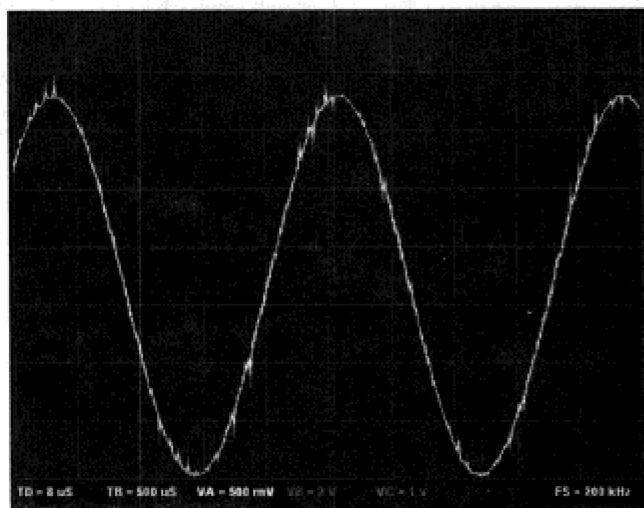


图 16-11　TestDA4 输出，440Hz 的正弦波

16.7　复制声音信息

一旦学会了如何生成语音，那就没有什么可以阻挡我们前进的脚步了。这个功能可以放入嵌入式控制领域中无限多的应用中去。用语音来提供反馈，用提醒和错误消息来引起用户的注意，或者处理恰当的话，也可以提高用户的使用经验，这样一来，任何与人交互的界面都可以得到很大的改善。但是，我们并不需要把自己局限在生成简单的音调或基本的旋律上。只要能够绘出所需波形，我们就可以复制各种声音。就像在前一个例子里使用的正弦函数表一样，我们可以使用一个更大的表，其中包含用专用仪器生成的完全正确的声音，甚至可以包含一条完整的语音信息。唯一的限制就是 PIC32 上的 Flash 程序存储器在存储应用程序之外，能够提供多少空间给数据表。

如果我们特别研究一下存储语音信息的可能性，知道人类声音的能量大部分集中在 400Hz

到 4kHz 的频率范围内，那么就能大大减少输出频率的需求，并将 PWM 按照每秒钟 8000 个样本的速率进行播放。但是还是应该保持较高的 PWM 频率，减少超出声音频率范围之外的 PWM 信号谐波，才能一直使用简单和便宜的低通滤波器。必须降低的只有 PWM 占空比的变化速率以及从表格中读取新数据的速度。例如，每 4 次中断才改变一次占空比，就能得到 11 025Hz 的采样速率。按照这个速度，理论上可以把存储在 PIC32MX360 的 Flash 存储器里的语音信息（8 位、单声道）播放长达 40 秒的时间。对于单芯片的应用来说，这个时间已经够长的了。

为了进一步提高（最好是能成倍提高）处理能力，我们可以寻找一些用于语音应用的简单压缩技术。比如 ADPCM 技术。ADPCM 表示*自适应差值脉码调制*（Adaptive Differential Pulse-Coded Modulation），它基于的前提是，两个连续样本之间的差值小于每个样本的绝对值，因此可以使用更少的位数对其进行编码。实际使用的位数得到优化，还能动态进行改变，从而在提供理想压缩比的同时，尽量减少信号失真。因此使用术语*自适应*。

16.8　媒体播放器

在本章的剩余部分，我们将探索一个更有雄心的工程。把在前面的几章中开发的所有库函数以及拥有的所有功能都用上，就可以创造一个基本的多媒体应用，可以播放存储在 SD/MMC 存储卡上的立体声音乐文件。

该应用需要从 PIC32MX360 上提供的 5 个 OC 模块中挑选出 2 个，另外基于输出声音质量的考虑，使用一个比单电阻电容电路（一阶低通滤波器）稍微高级一些的滤波器，到目前为止，我们在 TestDA 工程中一直使用的都是这种简单的滤波器。

使用诸如 MCP602 这样的低成本双运算放大器，我们可以设计一个非常简单的 Sallen Key（二阶）低通滤波器，适用于完全有能力驱动一副小型耳机或一个更强大的立体声放大器的音频频带（如图 16-12 所示）。

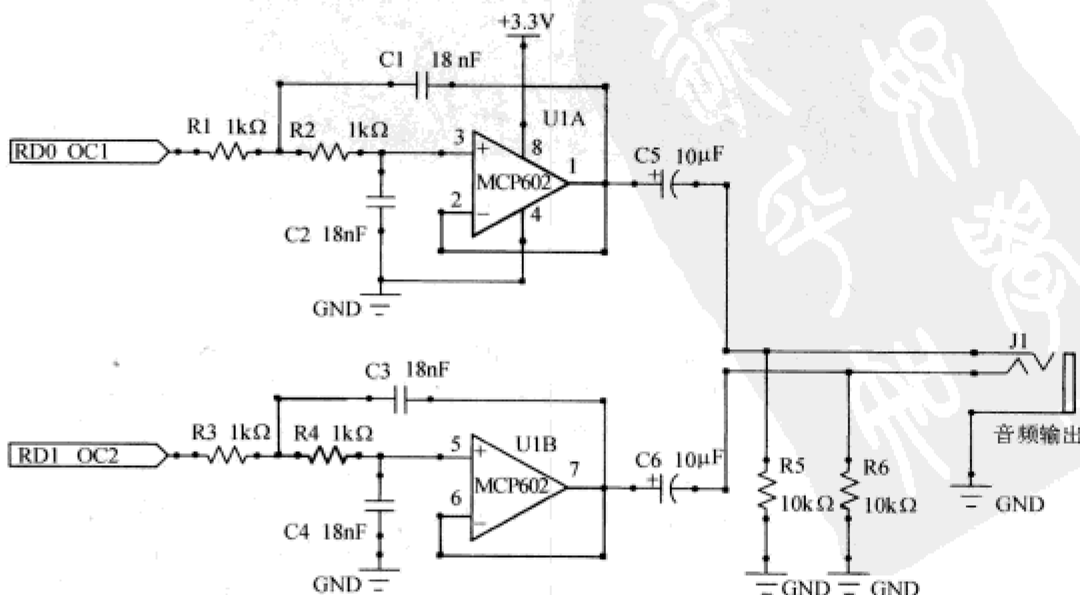


图 16-12　一个简单的音频 PWM 滤波器电路

至于所选的媒体格式，应该是未经压缩的 *WAVE* 格式，它和几乎所有的音频应用都兼容，从音乐 CD 中转换音乐文件时，WAVE 格式通常是默认的“无损”转换格式。

我们从创建一个全新的称为 Wave 的工程开始。现在就可以把 SD/MMC 底层接口（SDMMC.c）和访问 FAT16 文件系统的文件 I/O 库（fileio.c）加入到工程的源文件列表中。

16.9 WAVE 文件格式

在打开一个文件进行读取之后，我们需要理解数据编码的特定格式。带有.wav 扩展名的文件，用 WAVE 格式进行编码，是最简单、相关文档最齐全的一种文件。WAVE 格式是 *RIFF* 文件格式的变种，RIFF 文件格式是一个跨平台标准，使用特殊技术存储信息/数据的多个片段，把它们分成一个一个的块（chunk）。一个块（参见表 6-1）就是一个带有前缀的数据体，这个前缀包含 2 个 32 位的元素：块 ID 和块大小。

表 16-1 通用“块”格式

偏 移 量	大 小	描 述	值
0x00	4	块 ID	ASCII
0x04	4	块大小（内容的大小）	Size
0x08	Size	数据内容	
0x08 + Size	0~1	可选的填充数据	0x00

需要注意块的总大小必须是 2 的倍数，这样 RIFF 文件中的所有数据才可以按照字进行精确的对齐。如果数据块大小不是 2 的倍数，就需要填充一个额外的字节到块末尾。

存放 RIFF ID 的块通常位于 WAVE 文件的最开始，其数据块则以 4B 的类型字段开始。这个类型字段必须包含字符串 WAVE。块和块可以像俄罗斯套娃一样嵌套起来，不过在一个给定类型的块内部，也可以有多个子块。

表 16-2 阐述了 WAVE 文件中 RIFF 块的结构。

表 16-2 WAVE 类型的 RIFF 块

偏 移 量	大 小	描 述	值
0x00	4	RIFF 块 ID	RIFF
0x04	4	数据块大小+4	Size
0x08	4	类型 ID	WAVE
0x10	Size – 4	数据体（子块）	

数据体按照顺序包含一个 fmt 块和一个 data 块。一图胜千言，这次我们仍用图 16-13 来解释基本的 WAVE 文件布局。

fmt 块包含一个定义好的参数序列。该参数序列详细描述了包含在 data 块中的样本流，如表 16-3 所示。

在 fmt 和 data 块之间，可以有一些包含文件附加信息的其他块，所以我们可能得扫描块 ID，跳过一些块，直到找到想要找的（data）块为止。

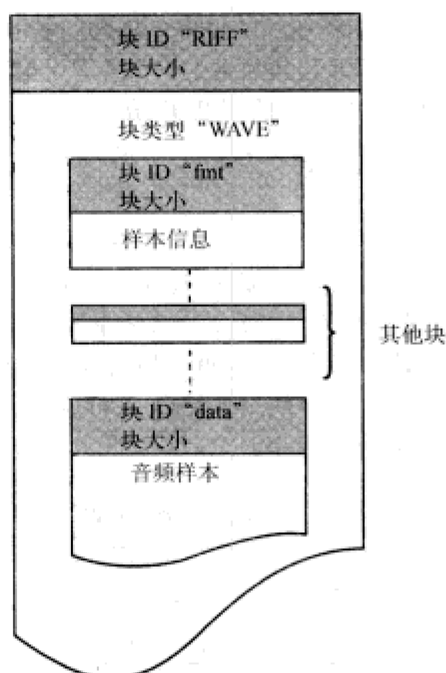


图 16-13　基本的 WAVE 文件布局

16.10　play()函数

创建一个新的 playWAV()函数，用于打开 WAVE 文件，并在获取并译码 fmt 块中的信息之后，配置两个 PWM 模块，把音频样本输入其中，复制一首完整的立体声歌曲。我们将把这个函数加入到 TestDA4.c 模块中，并重新命名为 AudioPWM.c。

表 16-3　fmt 块内容

偏移量	大小	描述	值
0x00	4	块 ID	Fmt
0x04	4	块大小	16+额外的格式字节
0x08	2	压缩码	无符号整数
0x0a	2	通道数目	无符号整数
0x0c	4	采样率	无符号长整数
0x10	4	每秒平均字节数	无符号长整数
0x14	2	块对齐	无符号整数
0x16	2	每个样本的有效位数	无符号整数（大于 1）
0x18	2	额外格式字节	无符号整数

```
/*
** AudioPWM.c
**
*/
#include <p32xxxx.h>
```

```
#include <plib.h>
#include <stdlib.h>
#include <explore.h>
#include <sdmmc.h>
#include <fileio.h>
#include "AudioPWM.h"

#define B_SIZE 512                    // audio buffer size

// audio configuration
typedef struct {
  char stereo;                        // 0 - mono 1- stereo
  char fix;                           // sign fix 0x00 8-bit, 0x80 16-bit
  char skip;                          // advance pointer to next sample
  char size;                          // sample size (8 or 16-bit)
} AudioCfg;

// chunk IDs
#define RIFF_DWORD      0x46464952UL
#define WAVE_DWORD      0x45564157UL
#define DATA_DWORD      0x61746164UL
#define FMT_DWORD       0x20746d66UL
#define WAV_DWORD       0x00564157UL


typedef struct {
  // data chunk
  unsigned int dlength;               // actual data size
  chardata[4];                        // "data"

  // format chunk
  unsigned  short     bitpsample;  // bit per sample
  unsigned  short     bpsample;    // bytes per sample
                                   // (4=16bit stereo)

  unsigned  int       bps;         // bytes per second
  unsigned  int       srate;       // sample rate in Hz
  unsigned  short     channels;    // # of channels
                                   // (1= mono,2= stereo)

  unsigned  short     subtype;     // always 01
  unsigned  int       flength;     // size of this block (16)
  char                fmt_[4];     // "fmt_"

  char type  [4];                  // file type name "WAVE"
  unsigned  int       tlength;     // size of encapsulated block
  char                riff[4];     // envelope "RIFF"
} WAVE;
```

WAVE 和 AudioCfg 数据结构用于收集所有的 fmt 参数，并把有用信息组织到一起；而块 ID 宏则用于识别不同的 ID，把这些 ID 当作 32 位整数，从而进行快速而高效的比较。

现在开始编写 playWAV() 函数。它只需要一个参数：文件名。

```
int playWAV( char *name)
{
  WAVE     wav;
  MFILE    *f;
  unsigned int lc, r;
```

```
    int wi, pos, rate, period, last;
    char s[16];

    // 1. open the file
    if ( (f = fopenM( name, "r")) == NULL)
    { // failed to open
      return FALSE;
    }
```

打开文件，如果失败则报错，否则立刻在数据缓冲区内寻找 RIFF 块 ID 以及 WAVE 类型 ID，这将证明我们是否读取到正确的文件：

```
  // 2. verify it is a RIFF formatted file
  if ( ReadL( f->buffer, 0) != RIFF_DWORD)
  {
    fcloseM( f);
    return FALSE;
  }

  // 3. look for the WAVE chunk signature
  if ( (ReadL( f->buffer, 8)) != WAVE_DWORD)
  {
    fcloseM( f);
    return FALSE;
  }
```

如果成功，则应该确认 fmt 块是数据体中的第一个子块。然后收集处理 data 块所需的所有信息，用于音乐回放。

```
  // 4. look for the chunk containing the wave format data
    if ( ReadL( f->buffer, 12) != FMT_DWORD)
    {
      fcloseM( f);
      return FALSE;
    }

  wav.channels   = ReadW( f->buffer, 22);
  wav.bitpsample = ReadW( f->buffer, 34);
  wav.srate      = ReadL( f->buffer, 24);
  wav.bps        = ReadL( f->buffer, 28);
  wav.bpsample   = ReadW( f->buffer, 32);
```

接下来，我们开始寻找 data 块，在 fmt 块之后的数据块中寻找块 ID 字段，如果不匹配，就跳过该块并继续寻找。

```
  // 5. search for the data chunk
  wi = 20 + ReadW( f->buffer, 16);
  while ( wi < 512)
  {
    if (ReadL( f->buffer, wi) == DATA_DWORD)
      break;
    wi += 8 + ReadW( f->buffer, wi+4);
  }
  if ( wi >= 512) // could not find in current sector
  {
    fcloseM( f);
    return FALSE;
  }
```

在寻找过程中，如果把当前缓冲区中加载的所有数据都找遍了也没有找到匹配，那就说明出问题了。



注解 通常情况下，从音乐 CD 中提取出的数据转换而成的.wav 文件中，在 fmt 块的后面紧跟着的就是 data 块。其他应用程序（例如 MIDI 接口）则可以生成包含更多复杂结构的 WAVE 文件，包括多个 data 块、播放列表、提示、标签等，但是我们的目标只是播放最基本形式的 WAVE 文件。

一旦找到匹配的 ID 字段，从 data 块的块大小字段就可以得知文件中包含的实际样本数量。

```
// 6. find the data size (actual wave content)
wav.dlength = ReadL( f->buffer, wi+4);
```

现在必须考虑播放的采样率，确定我们是否按原速进行播放。有可能发生所需采样率超出处理能力的情况，那么就不得不跳过一些样本以降低采样速率。我们把 48k 样本/秒作为上限，尽管严格说来，在高达 96k 样本/秒的速率下，PIC32 仍然能够产生 8 位分辨率的 PWM 输出。如果速率超过上限，就逐步除以 2，也就是逐步将跳读步长翻倍，直到速率符合要求为止。

```
// 7. if sample rate too high, skip
rate = wav.bps / wav.bpsample;        // rate = samples per second
ACfg.skip = wav.bpsample;             // skip to reduce bandwith
while ( rate > 48000)
{
  rate >>= 1;                         // divide sample rate by two
  ACfg.skip <<= 1;                    // multiply skip by two
}
```

接下来计算所需的 PWM 周期值（用于设置 PR2 寄存器）。如果所需周期超过了寄存器可以提供的位数（16 位），周期值就会超过 65 536，就会出现错误。

```
// 8. check if sample rate too low
period = (FPB/rate)-1;
if ( period > ( 65536L))              // max timer period 16 bit
{ // period too long
  fcloseM( f);
  return FALSE;
}
```

接下来，用一些参数对全局变量 ACfg 结构进行初始化，使中断服务例程可以管理音频的播放：

```
// 9. init the Audio state machine
CurBuf = 0;
pos = wi+8;                           // data begin
ACfg.stereo = (wav.channels == 2);
ACfg.size = 1;                        // #bytes per channel
ACfg.fix = 0;                         // sign fix / 16 bit file
if ( wav.bitpsample == 16)
{                                     // if 16-bit
  pos++;                              // add 1 to get the MSB
  ACfg.size = 2;                      // two bytes per sample
  ACfg.fix = 0x80;                    // fix the sign
}
```

在播放过程中，我们将记录从文件中提取出的样本数目，从而确定是否到达文件末尾。32 位的整型变量 lc 用于保存需要播放的剩余样本数目。

```
// 10 # of bytes composing the wav data chunk
lc = wav.dlength;
```

需要注意的是，我们到目前为止还没有使用 freadM() 函数；我们已经偷偷窥视了文件缓冲区内部的内容，知道 fopenM() 函数已经将它加载了。

为了使播放过程顺畅，我们使用双缓冲机制，在音频中断例程从一个缓冲区读取数据时，可以同时向另一个缓冲区注入新数据。数组 ABuffer[] 定义为 2 个块，每个块包含 B_SIZE 个字节（如图 16-14 所示）。

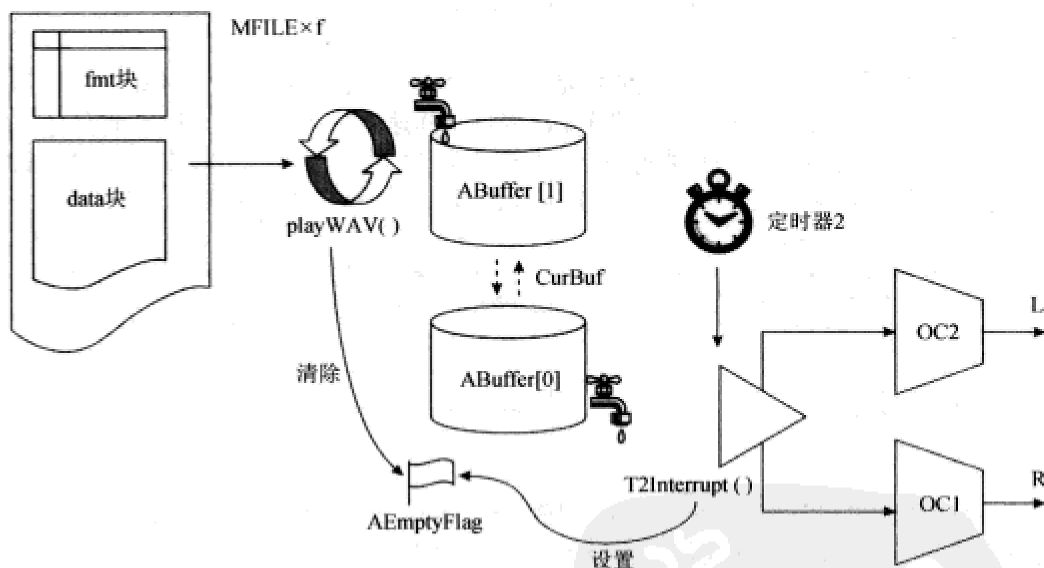


图 16-14　WAVE 播放器的数据流

为了获取最大性能，B_SIZE 应该选择为一个扇区的大小，或者是扇区大小的倍数，这样调用 freadM() 函数时，才能一次传递一整个扇区的数据。我们必须确保 freadM() 函数用于填充一个缓冲区的时间小于播放第二个缓冲区中所有数据的时间。在启动双缓冲机制之前，需要把两个缓冲区都填满：

```
// 11. pre-load both buffer
r = freadM( ABuffer[0], B_SIZE*2, f);
lc -= r;
AEmptyFlag = FALSE;
```

这时可以初始化“音频播放机”了，只需要改动 T2Interrupt() 函数，使用 OC1 和 OC2 模块，通过两个通道进行立体声播放。先调用 initAudio() 函数初始化 OC 模块，然后启动定时器 2 模块及其中断机制，从而启动播放过程。

```
// 12. configure Player state machine and start
initAudio();
startAudio( rate, pos, r-pos);
```

在定时器中断激活以后，服务例程立即开始处理来自于第一个缓冲区的数据，一旦该缓冲

区的所有数据都被处理完毕，就将 AEmptyFlag 标志位置位，表明新的数据必须从 WAVE 文件中读取，并且选择第二个缓冲区作为当前活跃缓冲区。因此，为了保持播放过程的顺畅进行，我们使用一小段循环，不断检查 AEmptyFlag 标志位，随时准备重新填充缓冲区、计算已经从文件中读取的字节数，直到整个文件处理完毕。

```
// 13. keep feeding the buffers in the playing loop
// as long as entire buffers can be filled
while (lc > 0)
{ // 13.1 check user input to stop playback
  if ( readKEY())                  // if any button pressed
  {
    lc = 0;                        // playback completed
    break;
  }
  // 13.2 check if a buffer needs a refill
  if ( AEmptyFlag)
  {
    r = freadM( ABuffer[1-CurBuf], B_SIZE, f);
    lc-= r;                   // decrement byte count
    AEmptyFlag = FALSE;       // refilled

  // 13.3 <<put here additional tasks>>
  putsLCD("\n");              // on the second line
  sprintf( s, "%dKB", (wav.dlength-lc)/1024);
  putsLCD( s);                // byte count
  }
} // while wav data available
```

在上述循环中，需要检查用户输入，读取 Explorer 16 演示板上按钮的状态，从而保证在任何时候，只要按下按钮就能停止音乐播放。在充满一个新的缓冲区之后，有一点点空余时间：这是处理附加（短小）任务的好时机，比如更新 LCD 显示器上的字节计数。

如果文件中剩下的数据不足以充满整个缓冲区，可以用最后一个样本的值把缓冲区的剩余位置填满。

```
// 14. pad the rest of the buffer
last = ABuffer[1-CurBuf][r-1];
while( r<B_SIZE)
  ABuffer[1-CurBuf][r++] = last;
AEmptyFlag = FALSE;          // refilled
```

一直等待，直到最后一个缓冲区的数据都被播放完毕，然后终止整个播放过程。

```
// 15.finish the last buffer
AEmptyFlag = FALSE;
while (!AEmptyFlag);

// 16. stop playback
haltAudio();
```

关闭文件，释放已分配的存储空间，返回调用程序。

```
  // 17. close the file
  fcloseM( f);

  // 18. return with success
  return TRUE;
} // play
```

16.11 音频例程

刚刚完成的 playWAV() 函数很大程度上依赖于底层音频函数来执行实际的定时器和 OC 外围设备的初始化工作，还包括周期性地更新 PWM 占空比。OC1 和 OC2 模块用来同步产生左声道和右声道的声音。定时器中断服务例程完成实际的核心播放功能，如同在前一个 TestDA 工程中一样。全局指针变量 BPtr 用于跟踪缓冲区中的当前位置，因为在每个周期都会把新的样本输入到 PWM 模块，直到所有数据用完。

```
void __ISR( _TIMER_2_VECTOR, ipl4) T2Interrupt( void)
{
  // 0. allow interrupt nesting
  asm( "ei");

  // 1. load the new samples for the next cycle
  OC1RS = 30+(*BPtr ^ ACfg.fix);
  if ( ACfg.stereo)
    OC2RS = 30 + (*(BPtr + ACfg.size) ^ ACfg.fix);
  else    // mono
    OC2RS = OC1RS;
```

注解 尽管可以给定时器 2 中断赋予中等优先级，但是我们希望能立刻重新使能中断，这样的话，具有更高优先级的中断就可以立即嵌套并服务。毕竟，我们可以奢侈地浪费充裕的采样周期时间（22μs，频率为 44.1kHz），用于更新 2 个 OC 模块的占空比，而其他高优先级中断（例如复合视频模块，前提是可以同时使用它）可能并不愿意等待直到这个中断处理完成。

指针向前移动的字节数取决于样本大小（每个 16 位或 8 位）；如果 playWAV() 函数确定需要降低采样率，指针向前移动的字节数也取决于需要跳过的样本数。

```
// 2. skip samples to reduce the bitrate
BPtr += ACfg.skip;
```

一旦整个缓冲区的数据都使用完毕，需要重新回到活跃缓冲区的头部。

```
// 3. check if buffer emptied
if ( --BCount == 0)
{
  // 3.1 swap buffers
  CurBuf = 1- CurBuf;

  // 3.2. place pointer on first sample
  BPtr = &ABuffer[ CurBuf][ACfg.size-1];

  // 3.3 restart counter
  BCount = B_SIZE/ACfg.skip;

  // 3.4 flag a new buffer needs to be filled
  AEmptyFlag = 1;
}
```

同时重新加载样本指针，复位样本计数器，设置标志位，告知 playWAV() 函数在用完当前数据之前需要把另一个缓冲区用新数据填满。这时才能清除中断标志，然后退出中断服务例程。

```
    // 4. clear interrupt flag and exit
    mT2ClearIntFlag();
} // T2Interrupt
```

初始化例程也和 TestDA 工程中的初始化例程有细微的不同。

```
void initAudio( void)
{ // configures peripherals for Audio playback
    // 1. activate the PWM modules
    // CH1 and CH2 in PWM mode, TMR2 based
    OpenOC1( OC_ON | OC_TIMER2_SRC | OC_PWM_FAULT_PIN_DISABLE,
            0, 0);
    OpenOC2( OC_ON | OC_TIMER2_SRC | OC_PWM_FAULT_PIN_DISABLE,
            0, 0);
    // 2. init the timebase
    // enable TMR2, prescale 1:1, internal clock, period
    OpenTimer2(T2_ON | T2_PS_1_1 | T2_SOURCE_INT, 0);
    mT2SetIntPriority( 4);    // set TMR2 interrupt priority

} // initAudio
```

实际的音频播放过程在使能定时器 2 中断之后才开始，同时必须保证播放状态机已经正确地初始化完毕：

```
void startAudio( int bitrate, int position, int count)
{ // begins the audio playback

    // 1. init pointers and flags
    CurBuf = 0;                 // buffer 0 active first
    BPtr = ABuffer[ CurBuf] + position;
    AEmptyFlag = FALSE;

    // 2. number of actual samples to be played
    BCount = count/ACfg.skip;

    // 3. set the period for the given bitrate
    PR2 = FPB / bitrate-1;

    // 4. enable the interrupt state machine
    mT2ClearIntFlag();          // clear interrupt flag
    mT2IntEnable( 1);           // enable TMR2 interrupt
} // startAudio
```

跟初始化相对应，haltAudio()函数则禁止定时器中断，从而停止输出比较模块的更新，同时停止整个状态机的运行。

```
void haltAudio( void)
{ // stops playback state machine
    mT2IntEnable( 0);
} // halt audio
```

为了完成音频模块，还需要一个简单的头文件，发布 playWAV()函数，这样工程的主模块才能使用该函数。

```
/*
** AudioPWM.h
*/
int playWAV( char *name);
```

16.12 一个简单的 WAVE 文件播放器

现在创建一个新的主模块，称为 WavePlayer.c。使用 LCD 显示器来提示用户，如果在播放前或者播放过程中出现错误，也在屏幕上进行显示（参见核心 playWAV()函数循环中的注释 13.3）。

```
/*
** WavePlayer.c
*/
// configuration bit settings, Fcy=72MHz, Fpb=36MHz
#pragma config POSCMOD=XT, FNOSC=PRIPLL
#pragma config FPLLIDIV=DIV_2, FPLLMUL=MUL_18, FPLLODIV=DIV_1
#pragma config FPBDIV=DIV_2, FWDTEN=OFF, CP=OFF, BWP=OFF

#include <p32xxxx.h>
#include <plib.h>
#include <explore.h>
#include <SDMMC.h>
#include <fileio.h>
#include <LCD.h>
#include "AudioPWM.h"

main( void)
{
  initEX16();
  initLCD();
  putsLCD( "Insert card...\n");
  while ( !getCD());
  Delayms( 100);

  if ( !mount())
    putsLCD("Mount Failed");
  else
  {
    clrLCD();
    putsLCD("Playing...");
    if (!playWAV( "VOLARE.WAV"))
    {
      clrLCD();
      putsLCD("File not found");
    }
  }

  while( 1)
  {
  } // main loop

} //main
```

生成工程，使用在线调试器对 Explorer 16 演示板进行编程，同时不要忘了给堆分配一些空间。因为 fileio 模块将把这些空间分配给缓冲区和相关数据结构（一定要多分配一些空间）。

为了使测试过程逐步进行，我推荐你在测试程序时，从采样率较低、文件体积较小的 WAVE 文件开始，然后逐渐提高采样率、增加文件大小。比如说，你应该用 8 位样本、单声道、8k 样本/秒的 WAVE 文件进行第一个测试。然后逐步增加格式的复杂性以及播放速度，最好在最后一个测试中，能够用 16 位样本、立体声、44 100 样本/秒的文件来测试本应用的全功能运行状

况。逐步递增这些要素的原因是，我们需要验证 fileio.c 模块的性能是否能够满足任务需求。随着采样率、声道数以及样本大小的增加，文件系统所需的带宽也在增加。我们可以快速计算出以上参数进行不同组合时所需的性能指标。

表 16-4 给出了每种文件格式所需的字节速率，也就是说，播放函数每秒钟需要处理的字节数目（样本大小×通道数目×采样率）。而最后一列则给出每隔多久，一个装满数据的新缓冲区需要再次被充满（512/字节速率），这也表明了 playWAV() 函数从 WAVE 文件中读取下一个扇区数据能花费的最长时间是多少。

表 16-4　WAVE 文件播放带宽需求

文　件	样本大小	通道	采样率	字节速率	重载周期（ms）
单声道声音	1	1	8 000	8 000	64.0
立体声声音	1	2	8 000	16 000	32.0
8 位单声道音频	1	1	22 050	22 050	23.2
8 位立体声音频	1	2	22 050	44 100	11.6
8 位高比特率单声道音频	1	1	44 100	44 100	11.6
8 位高比特率立体声音频	1	2	44 100	88 200	5.8
16 位单声道音频	2	1	44 100	88 200	5.8
16 位立体声音频	2	2	44 100	176 400	2.9

现在如果按照我的建议逐步开展实验，依照上表从上到下依次进行，就可以验证是否一直都可以流畅地播放任何类型的 WAVE 文件。到达最后一行时，它需要在不打断流畅播放的同时，能够一直保持超过 1.4Mbit/s（8 倍的字节速率）的比特率。

注解　既然我们为了简单起见而决定使用统一的 8 位分辨率的 PWM 输出，那么在播放表格中最后两种格式的 WAVE 文件时，就不能期待在音频输出质量上能有任何改善。这样做的结果只能是浪费 SD/MMC 存储卡的空间。如果想最大限度地利用现有的存储空间，就必须确保在把文件复制到存储卡上时，已经把样本大小减少为 8 位。这样就能在相同的存储空间里，存储两倍数量的音乐文件。

16.13　小结

最后这一章对于我们的"长途旅行"来说，算是一个理想的终点，因为我们把很多先进的软硬件功能都集成到了一个工程里，并同时覆盖了数字领域和模拟领域。我们使用输出比较模块来产生声音频谱范围内的模拟信号，并把这个新功能和前一课里开发的 fileio.c 模块结合起来，实现了从大容量存储设备（SD/MMC 卡）中播放无压缩声音文件（WAVE 文件格式）的功能。我们开发的基本媒体播放器只是一个新的开始。这个工程的功能扩展可以说是无限的，并且如果我已经激起了你的好奇心和想象力，那么对于整个 PIC32 和 MPLAB C32 编译器来说，其使用方法也是无限的。

16.14　提示与技巧

对于 PWM 模块来说，播放的开始与结束是两个非常关键的部位。不工作时输出滤波器电

容是放电状态，输出电压则是 0V。但是一旦播放开始，50%的占空比就会迫使它快速上升到大约 1.5V 的电平处，产生很大很刺耳的噪声。而在播放结束时则情况正好相反，因此我们在关闭 PWM 模块时不能像在示例工程中那样只禁止中断。但这种现象和模拟放大器电路在上电和关闭时也没有什么不同。加入几行代码可以解决这个问题。在定时器中断使能和播放机启动前，加入一个小（定时）循环来逐步增加输出占空比，从零一直上升到取自播放缓冲区的第一个样本值为止。

16.15 练习

（1）调研 ADPCM 译码在语音信息中的使用（见使用说明书 AN643）。

（2）搜索存储卡上的所有.wav 文件并建立一个播放列表。

（3）使用伪随机数生成器和播放列表，实现随机播放模式。

（4）执行实时的信号频谱分析（FFT）并用视频动画显示结果（采用图形均衡器的显示方式）。

16.16 参考书

Dino Mandrioli 和 Carlo Ghezzi 所著的 *Theoretical Foundation of Computer Science*。这本书比较深奥，但是如果你对计算机科学所基于的数学理论基础感兴趣，就可以钻研一下该书。

16.17 链接

http://en.wikipedia.org/wiki/RIFF。阐述了什么是 RIFF 文件格式。

http://en.wikipedia.org/wiki/WAV。阐述了什么是 WAVE 文件格式。

http://ccrma.stanford.edu/courses/422/projects/WaveFormat/。对 WAVE 文件格式的另一种很好的阐述。

16.18 免责声明

不要在家做这些实验！

16.19 对于一些行家的最后提示

意大利流行音乐之父多明戈 · 莫都格诺曾在“Nel Blu Dipinto di Blu”[①]中唱道：梦想把自己的脸和手都涂成蓝色，然后被突然刮来的一阵风托起，飞向蓝色的天空。

勇敢地让自己的梦想成真吧！

① 该歌曲中文译名为《蓝天飞翔》。——译者注

Programming 32-bit Microcontrollers in C
Exploring the PIC32

32位单片机C语言编程 基于PIC32

嵌入式控制解决方案全球领导厂商Microchip公司推出的32位微处理器PIC32采用MIPS内核，具有优异的速度和性能，并且能无缝移植基于8位和16位架构微处理器的应用程序，适应了嵌入式系统微型化、智能化的发展方向。

本书依托PIC32平台，详细介绍了基于C语言的嵌入式控制系统的软件设计方法。全书从基础知识入手，循序渐进，使读者快速了解嵌入式控制系统软件的架构，然后通过精心设计的实例展示PIC32的各种片上外围设备，最后用新颖的、趣味性极强的扩展内容介绍PS/2键盘控制、视频显示、MMC/SD卡接口以及音频处理等技术。

无论是8位还是16位嵌入式系统设计人员，只要具备基本的C语言编程知识，都能通过本书轻松掌握PIC32架构，并从汇编语言程序设计高手成功转型为C语言编程高手！

Lucio Di Jasio 嵌入式控制系统设计专家，在PIC架构设计方面具有丰富的经验。曾任职于Microchip公司，对其产品性能以及开发流程都非常熟悉。除了本书外，他还著有《16位单片机C语言编程：基于PIC24》一书。



延伸阅读

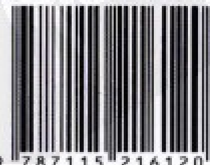
- 嵌入式系统设计的艺术（第2版·英文版） 978-7-115-19521-0 49.00元
- PIC技术宝典 978-7-115-18554-9 99.00元
- PIC嵌入式系统开发 978-7-115-18265-4 69.00元
- 8051微控制器（第4版） 978-7-115-17959-3 49.00元
- 16位单片机C语言开发：基于PIC24（即将出版） Lucio Di Jasio 著
- 8位单片机C语言开发：基于PIC16（即将出版） Martin P. Bates 著

本书译自原版*Programming 32-bit Microcontrollers in C: Exploring the PIC32*，并由Elsevier授权出版。









ISBN 978-7-115-21612-0
定价：49.00元

本书相关信息请访问：图灵网站 http://www.turingbook.com
读者/作者热线：(010) 51095186
反馈/投稿/推荐信箱：contact@turingbook.com

分类建议：电子电气/单片机
人民邮电出版社网址 www.ptpress.com.cn